RUHR UNIVERSITÄT BOCHUM

RUB

Open C3S

Open Competence Center for Cyber Security

Cryptanalytic Tools

Authors:

Prof. Dr.-Ing. Tim Güneysu

Dipl. Ing. Alexander Wild

B. Sc. Tobias Schneider

Ruhr-Universität Bochum

Module
Cryptanalytic Tools

Authors:
Prof. Dr.-Ing. Tim Güneysu
Dipl. Ing. Alexander Wild
B. Sc. Tobias Schneider

---

1. edition

Ruhr-Universität Bochum

Universitätsstraße 150
44801 Bochum

1. edition (31. March 2015)

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 16OH12026 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Contents

Introduction to the module books

I. Icons and colour codes

Example	B
Definition	D
Theorem	S
Exercise	Ü

Chapter 3 Introduction to Cryptanalysis

In this chapter you will learn the basic concepts behind the foundation of security and cryptographic systems, namely the concept of cryptanalysis, attacker characteristics and cost models.

3.1 Definition of Security

Modern IT-security that is designed to satisfy system security goals for many years cannot exist without involvement of cryptographic systems. With all security guarantees founded on such cryptosystem, we are in need for rigorous *security proofs* to backup any claims on security. Unfortunately, security proofs can be provided for hardly any practical security system due to the inherent complexity of the cryptographic structures involved. Therefore we found our confidence in today's cryptosystem by using techniques from *cryptanalysis*. However, cryptanalysis is solely based on observations and identified weaknesses of a security assumptions, hence we are faced with the following problems:

- *Bounds of Cryptanalysis:* Given that we currently assume to know an cryptanalytic attack, are there still better algorithms to cryptanalyze a given cryptosystem?
- *Inputs to Cryptanalysis:* Given that we have a best known cryptanalytic attack, what are optimal input parameters to achieve best results (e.g., finding optimal differential trails to identify a collision within a cryptographic hash function or block cipher)?
- *Practicality of Cryptanalysis:* Given we have identified a best cryptanalytic attack, what are the actual implementation cost and which is the optimal platform to run the attack – if feasible at all?

The nature of the aforementioned issues obviously allows only for empiric solutions. To evaluate such solutions, we require cryptanalytic tools implemented on a range of platforms to practically evaluate and identify the potential of each individual attack. This is the primary goal of this lecture, to investigate and elaborate on (generic) strategies and tools to cryptanalyze cryptographic systems used in practice today.

3.1.1 Security of Cryptographic Systems

Before we delve into details of cryptanalysis, we should define the notion of security with respect to cryptographic systems. A fundamental principle in cryptography was defined by Auguste Kerckhoff in 1883.

Definition 3.1: Kerckhoffs's Principle (Kerckhoff [1883])

A cryptosystem should be secure even if everything about the system **except the key** is public knowledge.

D

In other words, it is considered bad practice when the security of a system relies on the secrecy of its internal structures that means it relies on the fact that cryptanalysis cannot be conducted due to the lack of information of the attacker (cf. *security though obscurity*). History has shown that an attacker can always acquire at least some information about the cryptographic internals at some point in time. If this information is revealed from the design point of view, this directly corresponds to

losing parts of the secret and a (partial) break of the system. Hence, it is preferable to declare everything as public information and only a very small part of the cryptographic system as secret information which is typically referred to as *secret parameter* or *secret key*. This reduces the effort for the designer to only protect this secret key at all costs.

3.1.2 Categories of Attacks

To recover this secret key, many different kinds of cryptanalytic techniques have been used over the last years. The following attacks can be found against the keys of encryption systems:

Distinguishing Attacks: Any form of cryptanalysis on encrypted data that allows an attacker to distinguish the encrypted data from random data (e.g., statistical tools)

Related Key Attacks: Any form of cryptanalysis where the attacker can observe the encryption system under different but related keys, where the relation is chosen by the attacker.

Full Message Recovery: Any form of cryptanalysis where the attacker can recover the plaintext without the key.

Full Key Recovery: Any form of cryptanalysis that is able to recover the secret that was used to encrypt a certain message.

Besides different scopes of key attacks, there are also different views on the potential of the attackers. A strong encryption system needs to be secure even if an attackers has the capabilities and the access to interact with the cryptographic system in any possible way. We differentiate the attacker's capability and the following attack scenarios on encryption systems:

Black Box: Contrary what is stated by Kerckhoff's principle the attacker has no information about the inner workings of the system. Only the input and the output of the system can be observed by the attacker to reveal information about the internal secrets.

Ciphertext-Only: The attacker has access to all algorithmic details about the involved cryptosystem as well as multiple ciphertexts. However, he has no access to corresponding plaintexts or any other secret-key related information. The attacker can now either attempt to recover the secret key used to generate the ciphertext or – alternatively – he just attempts to identify corresponding plaintexts without actually possessing the secret key.

Known-Plaintext: The attacker knows several plaintext-ciphertext pairs encrypted under a single secret key using a known cryptosystem.

Chosen-Plaintext: The attacker has access to an encryption oracle, i.e. the attacker is able to choose a plaintext and send it to a third party or closed instance that feeds this input to the cryptosystem for encryption. The attacker is then provided with the resulting encrypted plaintext.

Chosen-Ciphertext (adaptive): Same as Chosen-Plaintext but now the oracle decrypts plaintext. In the adaptive setting, the attacker can request multiple decryptions from the oracle and is allowed to adapt the next decryption input by additional information gathered from previous rounds.

Note that cryptography is not solely for use to provide confidentiality using encryption systems. Likewise cryptography is widely deployed to provide authentication

and integrity of information, also known as signatures (based on *asymmetric cryptography*) or authentication codes (based on *symmetric cryptography*). For further information concerning symmetric and asymmetric cryptography please refer to (Paar and Pelzl [2010]). For such authentication systems, an attacker might have different attack goals to forge a valid signature s to a given m defined as follows:

Existential Forgery: The attacker attempts to forge a valid signature s without any control over the message m. Although the inability to control the message seems like an esoteric target for an attacker, it still can be very useful to fool initial automated verification mechanisms to complete subsequent attack procedures.

Selective Forgery: The attacker can create a valid signature s_0 for a very specific given message m_0.

Universal Forgery: Even without knowing the private key, the attacker can create valid signatures s_i to arbitrary messages m_i.

Total break: The attacker can compute the private key and completely impersonate the target.

Similarly to the attacks on the encryption systems we distinguish different attackers for authentication systems involving asymmetric cryptography. Given access to different items relevant for an attack, attackers can be classified as follows:

Public-Key Only: The attacker has solely access to the public key from which he tries to compute the private key.

Known Signature Attack: The attacker has access to the public key and a valid signed message. The attacker's target for this setting is either to identify the private key or to create another valid signature for a second message.

Chosen Message Attack: The attacker has access to a signing oracle and can choose multiple messages $m_0, \ldots, m_i$ for which signatures will be generated. Target for the attacker is to acquire sufficient information from the previous signing attempts on $m_0, \ldots, m_i$ so that he is finally able to generate a valid signature for a message m_{i+1} (that has not been signed by the oracle before) on his own.

In light of the aforementioned attack options one usually cares about how long it will take an attacker to circumvent the security and what resources he needs in order to have a reasonable chance for success. This cost is typically measured in time and/or money and its sum must be higher than the value (in a similar measure) of the target asset. For instance, if a one million dollar machine is needed to obtain secrets of value one dollar, one could hope the attacker is deterred by this (unless he/she simultaneously can get one million such secrets of course), and similarly, if information is only sensitive for a few hours, a break of the system requiring a week's attack effort *might* be acceptable.

3.1.3 Categories of Attackers

In the same way, the security designer needs to take into account the potential of an attacker. In 1996, (Blaze et al. [Jan. 1996]) used the following classification to distinguish the financial resources for attackers. Note that in this context, special-purpose hardware (such as FPGA or ASIC system) play a major role to classify the strength of such attackers to attack an individual cryptographic target.

"Hacker": using a $0 budget using one or multiple standard PCs, or possibly a few $100 spent on FPGA hardware.

Table 3.1: Minimum symmetric key-size in bits for various attackers.

Attacker	Budget	Hardware	Min security (bit)
"Hacker"	0	PC	53
	< $400	PC(s)/FPGA	58
	0	"Malware"	62
Small organization	$10k	PC(s)/FPGA	64
Medium organization	$300k	FPGA/ASIC	68
Large organization	$10M	FPGA/ASIC	78
Intelligence agency	$300M	ASIC	84

Table 3.2: Key-size equivalence between different cryptosystems.

Security (bits)	RSA	DLOG		EC
		field size	subfield	
48	480	480	96	96
56	640	640	112	112
64	816	816	128	128
80	1248	1248	160	160
112	2432	2432	224	224
128	3248	3248	256	256
160	5312	5312	320	320
192	7936	7936	384	384
256	15424	15424	512	512

Small organization: with a $10k budget using one or multiple FPGAs.

Medium organization: $300k budget using FPGAs and/or ASICs.

Large organization: $10M budget using FPGAs/ASICs.

Intelligence agency: $300M budget using ASICs.

Such a classification is probably largely still valid today considering a single attack case[1]. Taking this into account, we now need to define how to choose secret keys accordingly to withstand attacks against these types of attackers.

3.1.4 Secret Key Lengths

The general approach for choosing security keys is to find the best trade-off between the security level against the highest projected attacker type and the efficiency of the cryptosystem with respect to its implementation complexity and runtime on the target platform. Table 3.1 (obtained from (ECRYPT II - European Network of Excellence in Cryptology II [(2011-2012])) shows the minimum key length in bit for different security levels according to the attacker classification shown above.

Note that these are *minimum* sizes, giving protection only for a few months. For a higher level of protection, Table 3.2 provides additional information for different cryptosystems. Given any desired (symmetric key) security level, one may need to convert the symmetric key into an equivalent asymmetric key size for cryptosystems such as RSA, discrete logarithm-based (DLOG) or elliptic curve-based (EC).

We have not introduced any details considering the metric used for determining the key length. Generally, the key lengths are computed according to the asymptotic complexity of the best known attack. We will see in Chapter II what asymptotic

[1] For 2013 it was reported that the NSA has a budget of 11 billion US$ to spend on spying and cryptanalytic activities.

complexity exactly means. Roughly spoken, it is the close-up on a one-dimensional feature of the respective attacking algorithm that is mapped to a "'similar"' class of functions. In other words this complexity metric has been heavily truncated in its precision as it just roughly counts a single type of computation steps, e.g., number of logical bit operations or multiplications. Even worse, the definition of the computation step can differ between different attacking algorithms, i.e., the individual step for attacking a symmetric cipher is usually by far less complex than the step to attack the factorization of RSA. For a fair comparison, all these steps need to be compared, e.g., by implementation on the same platform. Next the implemented step can be used to extrapolate the practical cost for the full attack. The implementation on the platform providing the best cost-performance ratio finally determines the actual cost for the attack. In particular, this includes additional implementation-specific metrics such as:

- Absolute runtime and runtime in cycles
- Memory size requirement
- Memory access time
- Chip and circuit cost
- Communication cost and latency
- Energy cost

Cryptanalytic tools are implementations of cryptanalytic algorithms that support the practical evaluation of attacks over a range of different computing platforms. This will be the primary target for our investigations.

3.2 Outline of this Lecture

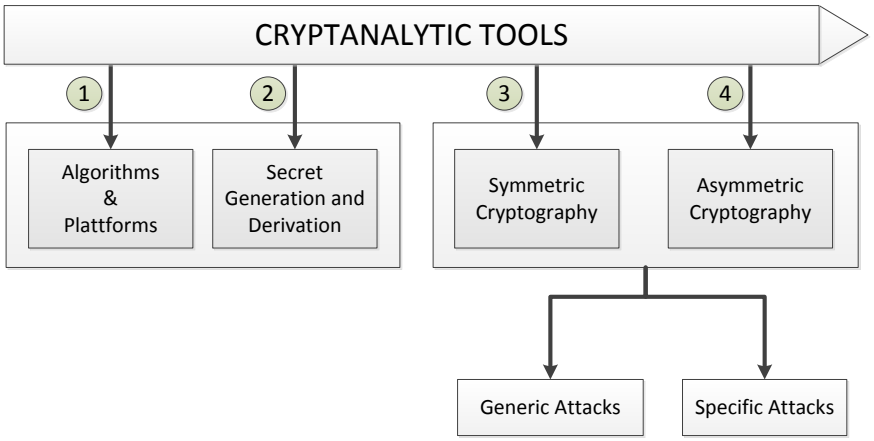
In the course of this lecture we will attempt to reveal the secrets of symmetric and asymmetric cryptographic algorithms using a series of different cryptanalytic algorithms and tools. Therefore, we will work on some preliminaries first to understand the basic concepts and problems. This includes the asymptotic complexity of algorithms but also their structural implementation on computation platforms as well as possibilities for their parallelization. Next, we identify how to generate or derive secrets for use in typical cryptographic systems. In particular, we will have a look at straight attacks that solely target the generation and derivation process of the secret without taking any cryptographic instance into account. Then, we will cover a broad range of cryptanalytic attacks and key searching methods, encompassing both the field of asymmetric as well as symmetric cryptography. In this context, we will distinguish between generic and specific attacks. While we mostly focus on generic attacks since they can be widely applied to all cryptosystems of a class, we selectively also investigate some specific attacks that require special properties of the underlying cryptosystems or inputs to succeed.

A graphical overview of the contents of this lecture can be found in Figure 3.1.

3.3 Further Reading Materials

This lecture covers a topic that requires a vast variety of knowledge and interdisciplinary skills, ranging from the mathematical understanding of cryptanalytic properties to parallel computation issues on special-purpose hardware. Obviously, there exist no resource reference that can holistically treat all relevant aspects of this lecture. However, the following books and reading materials can help to understand the background on the techniques which are in particular beyond the scope of these lecture materials.

Figure 3.1: Overview of the lecture contents



- *Cryptography:* A basis work for cryptography and cryptographic algorithms is the *Handbook of Applied Cryptography* (Menezes et al. [1996]), in which a great number of general algorithms and methods is covered. More literature on specific topics can be found in other locations and sources, e.g. the book of the authors of the Advanced Encryption Standard (AES) "'*The Design of Rijndael*"' (Daemen and Rijmen [2002]) in which the background of the algorithm is covered, too. A short introduction to the RSA cryptosystem can be found in (Coutinho [1999]) and more knowledge about number theory can be found in (Koblitz [1994]). A very good source for Elliptic Curve Cryptography is (Hankerson et al. [2004]), a more mathematical approach is chosen by (Avanzi et al. [2005]). Finally, a good resource for practitioners in cryptography is also (Paar and Pelzl [2010]).
- *Symmetric Cryptanalysis:* Extensive background materials on the design and cryptanalysis of symmetric block ciphers can be found in the book by Knudson and Robshaw (Knudsen and Robshaw [2011]).
- *Asymmetric Cryptanalysis:* A good overview of cryptanalytic algorithms in the domain of asymmetric cryptography was published by Nguyen (Nguyen [2008]).
- *Implementation and Processor Design:* Skills for general processor design and optimization can be acquired in the books of Wuest (Wüst [2009]) and Bringschulte and Ungerer (Brinkschulte and Ungerer [2007]).

Chapter 4 Computational Complexity and Parallelism

In this chapter you will learn the basic concepts of asymptotic complexities, computation models and palatalization methods.

4.1 Asymptotic Computational Complexity

Algorithms require a number of steps to solve a specific problem. This number is usually denoted as the running time of the algorithm. The notion of a step needs to be regarded with respect to the underlying machine model and is assumed to be executed in constant time. Generally speaking, the running time depends on the size of the problem and on the respective input.

In order to evaluate an algorithm independently of the input, the notion of time complexity is introduced. The time complexity $T(n)$ is a function of the problem size n. The value of $T(n)$ is the running time of the algorithm in the worst case, i.e. the number of steps it requires at most with an arbitrary input. In addition to this notion, sometimes the behavior in the average case is considered, i.e. the mean number of steps required with a large number of random inputs.

We regard the running time of an algorithm as a function of input size n, using an abstraction for the complexity *for large n*. This is done by using special notation describing the *asymptotic computational complexity* of an algorithm in relation to a reference functions. The symbols used for this abstraction are the so called Landau symbols known as $\Theta, O, \Omega, o, \omega$.

4.1.1 Big Θ-Notation

The set of all functions that have the same rate of growth as $g(n)$. That means that $g(n)$ is an *asymptotically tight bound* for $f(n)$ starting from n_0.

Definition 4.1: $\Theta(g(n))$

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:
$\Theta(g(n)) = \{f(n) \exists c_1, c_2, n_0 \in \mathbb{N}^+$ such that $\forall n \geq n_0$,
we have $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$

D

Example 4.1: $t(n) = 10n^2 - 3n = \Theta(n^2)$

Obviously the highest-order term is predominant in describing the runtime of the algorithm $T(n)$. For an exact definition, we need to determine suitable constants for n_0, c_1, c_2. The typical strategy for this is choosing c_1 a little smaller than the leading coefficient of the highest-order term, and c_2 a little bigger:
$c_1 = 9, c_2 = 11, n_0 = max\{i \in \mathbb{N} : 9 \cdot i^2 \leq f(n), f(n) \leq 11 \cdot i^2\} = 3.$

B

4.1.2 Big O-Notation

The set of all functions whose rate of growth is the same as or lower than that of $g(n)$. That means that $g(n)$ is an *asymptotically upper bound* for $f(n)$. You can

compare this description with the Figure 4.1. We can see that all values of $f(n)$ from the point n_0 on, are smaller (or equal) to the upper bound described by $c \cdot g(n)$.

D

Definition 4.2: $O(g(n))$

For function $g(n)$, we define $O(g(n))$, big-Oh of n, as the set:
$O(g(n)) = \{f(n) \exists c, n_0 \in \mathbb{N}^+$ such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq c \cdot g(n)\}$

4.1.3 Big Ω-Notation

The set of all functions whose rate of growth is the same as or higher than that of $g(n)$. That means that $g(n)$ is an *asymptotic lower bound* for $f(n)$. Figure 4.1 shows this behaviour. We can see that all values of $f(n)$ from the point n_0 on, are bigger (or equal) to the lower bound described by $c \cdot g(n)$.

D

Definition 4.3: $\Omega(g(n))$

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:
$\Omega(g(n)) = \{f(n) \exists c, n_0 \in \mathbb{N}^+$ such that $\forall n \geq n_0$,
we have $0 \leq c \cdot g(n) \leq f(n)\}$

4.1.4 Little o-Notation

D

Definition 4.4: $o(g(n))$

For function $g(n)$, we define $o(g(n))$, little-oh of n, as the set:
$o(g(n)) = \{f(n) \exists c, n_0 \in \mathbb{N}^+$ such that $\forall n \geq n_0$,
we have $0 \leq f(n) < c \cdot g(n)\}$

That means $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity: $\lim_{n\to\infty}[f(n)/g(n)] = 0$. Also $g(n)$ is an *upper bound* for $f(n)$ that is not asymptotically tight.

4.1.5 Little ω-Notation

D

Definition 4.5: $\omega(g(n))$

For function $g(n)$, we define $\omega(g(n))$, little-omega of n, as the set:
$\omega(g(n)) = \{f(n) \exists c, n_0 \in \mathbb{N}^+$ such that $\forall n \geq n_0$,
we have $0 \leq c \cdot g(n) < f(n)\}$

That means $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity: $\lim_{n\to\infty}[f(n)/g(n)] = \infty$. Also $g(n)$ is a *lower bound* for $f(n)$ that is not asymptotically tight.

4.1.6 Algorithmic Runtimes

The *runtime* $O(f(n))$ means that the runtime is $O(f(n))$ in a worst-case scenario and is therefore the bound on the running time of every input. This extends to $\Theta(f(n))$ as well. The Theta-notation is also bounded by the worst-case running time and

therefore a bound on the runtime of every input. On the other hand, the Omega-notation provides a notion for the best-case runtime with $\Omega(f(n))$. In particular the relation $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ means that we can obtain asymptotically tight bounds from the asymptotic upper and lower bounds $O(g(n)$ and $\Omega(g(n))$, respectively.

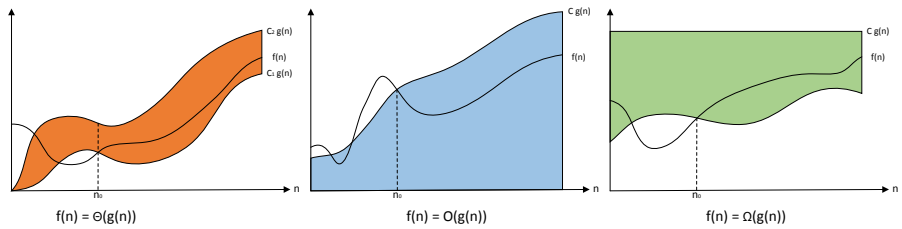


Figure 4.1: Relations between the different complexities

We now like to give two examples for using Landau symbols for algorithm runtime determination, one from an algorithmic and one from a mathematical point of view:

Example 4.2: Complexity of Insertion Sort

Insertion sort takes $\Theta(n^2)$ in the worst case, and sorting (as a problem) $O(n^2)$. We can explain this, because in insertion sort the algorithm has to run through the whole array of elements in order to check for the existence each item, and then rerun the array (in a worst case scenario) to find its correct position. In fact, other smarter sorting algorithms (e.g., *merge sort*) can reduce the runtime of sorting to $\Theta(n \cdot \log n)$.

B

Next we introduce an example which demonstrates how to compute and simplify terms into an asymptotic complexity expression.

Example 4.3: Simplifying Asymptotic Complexities

$4n^3 + 3n^2 + 2n + 1$
$= 4n^3 + 3n^2 + \Theta(n)$
$= 4n^3 + \Theta(n^2)$
$= \Theta(n^3)$.
That means we can treat $\Theta(f(n))$ in equations as an *anonymous function* $g(n) \in \Theta(f(n))$. In this examples for instance, the term $\Theta(n^2)$ represents $3n^2 + 2n + 1$.

B

4.2 Computation Platforms

Algorithms are procedures to process and transform given inputs into outputs. For execution of algorithms, a computation platform is required. The processing platform is characterized by its processor datapath, feature and instruction set, memory architecture, parallelism capabilities and software package.

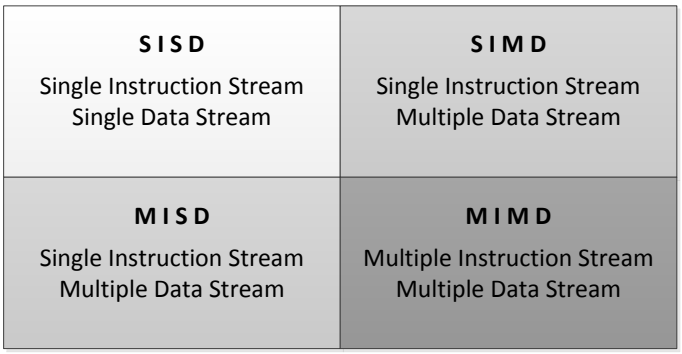
The *processor data path* is usually defined by the width of the processed data per instruction. Typical sizes for data paths of contemporary processing platforms are 8, 16, 32 and 64 bits of data (Brinkschulte and Ungerer [2007]). The performance is determined by the throughput and latency for each instruction which depend on the microarchitecture of the processor. In particular for long processing pipelines, high instruction latency needs to be taken into account.

The *processor feature and instruction set* is defined by the type of functional units and instructions available in the processor core. For example, a processor can natively support $n \times n$ bit multiplication by a single instruction or it can leave this task for implementation by the developer (who then is required to either implement an own functions based on an $m \times m$ bit multiplication with $m < n$ or a fully serial shift-and-add software multiplication for this purpose). Typical features of modern processors are floating-point capabilities, a dedicated division unit or cryptographic support for the AES block cipher and random number generation. Further characteristics of a processor are the use and length of the processor pipeline, out-of-order instruction scheduling, super-scalarity and multi-operation instruction coding techniques (e.g., Very-Long-Instruction-Word (VLIW) (Fisher [1983]) or Explicitly Parallel Instruction Computing (EPIC) (Schlansker and Rau [2000])).

Processing data requires the data to be stored before and after the operation. For this a *hierarchy of different memories* is used to provide quick and volatile access to data for immediate processing (using Caches, Scratch-Pads, SRAM/DRAM/S-DRAM) or long-term non-volatile purposes (e.g., flash and hard disk drives, optical media or tapes). The latency and throughput required by the processor to access or store a piece of data through the memory hierarchy directly determines the overall performance of the algorithm.

In most modern processors, more than just one functional core are present to accelerate computations by means of parallelism (Kumar [2002]). Parallelism exists both for the instruction *and* data stream. M. Flynn (Flynn [1972]) proposed the classification of parallel computation units as depicted in Figure 4.2.

Figure 4.2: Instruction and Data Stream Parallelism



This classification of parallelism can be applied on two different levels of the computation hierarchy:

- *Micro-Level Parallelism*: Multiple Instructions can be executed in the same clock cycle by independent processing cores (MIMD), e.g., using VLIW techniques or super scalarity. In processors populated with vectorization units, single instruction multiple data instructions (SIMD) can be used to process a vector with the same operation simultaneously. Additionally, inherent parallelism inside instruction can be achieved using the processor features explained above (super scalarity, instruction pipelining)
- *Macro-Level Parallelism*: Macro parallelism is the property of a parallel algorithms to run different computations on different sets of data. This is typically supported by multiple processor cores in either an homogeneous or heterogeneous processor setting. For example, in hybrid processor architectures a central computing core coordinates the work among multiple

supporting cores. IBM's Cell processor consists of a central PowerPC that provides processing data to 7-8 Synergistic Processing Elements (SPE) over a high-speed ring interconnect (Gschwind et al. [2006]). In contrary, multi-core architectures feature many identical cores that can share one or work on different tasks as required.

Parallel computing systems need special consideration with respect to other properties, in particular access to the memory hierarchy. For many parallel applications and algorithms, it is crucial to precisely define the requirements (i.e., data bandwidth, size and access frequency) for each individual process to identify the best memory hierarchy and interconnect. Figure 4.3 shows different strategies to realize access of parallel processors (P) to the same and different portions of memory (M).

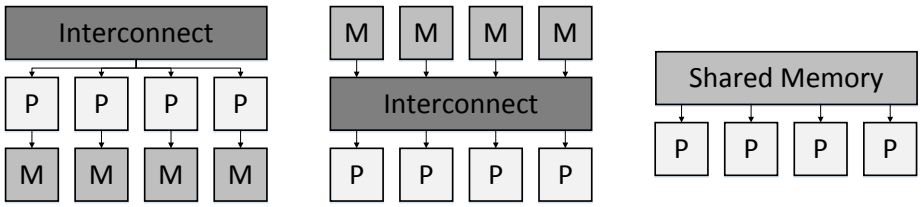


Figure 4.3: Processor-Memory Configurations and Interconnects

The more specialized a computing platform is the more complex implementations will become. On the one hand general-purpose processors (GPP) provide a good mix of features for many applications. Based on compiler-assisted software development including a wide range of existing prebuilt libraries, complex algorithms can be realized without high development efforts. However, the high abstraction and versatility of these platforms make them very inefficient due to lack of specialization. This can be achieved by Application-Specific Instruction-Set Processors (ASIP) as well as hardware-specific implementation on FPGAs (Field-Programmable Gate Array) and ASICs (Application-Specific Integrated Circuit). Figure 4.4 shows the difference in computational efficiency and the development cost of different platforms.

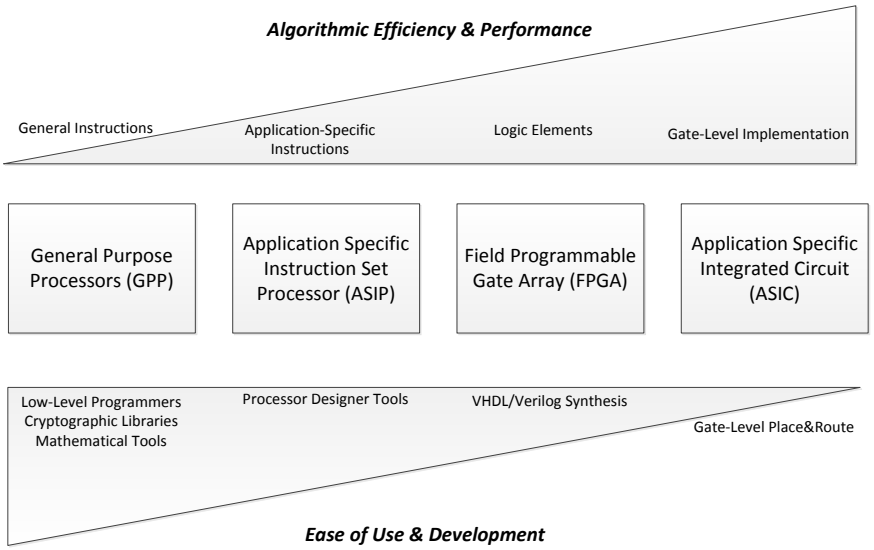


Figure 4.4: Overview of Different Processor Types

4.3 Parallel Computation Models

In light of the large implementation specific differences of computation platforms, it seems helpful to define an abstract model for solving problems algorithmically first before optimizing algorithms towards a specific platform. The designer of a sequential algorithm typically formulates the algorithm using the abstract model of a random-access machine (RAM). In this model, the machine consists of a single processor connected to a memory system. Each basic CPU operation, including arithmetic operations, logical operations, and memory accesses, requires one time step. The designer's goal is to develop an algorithm with modest time and memory requirements. The random-access machine model allows the algorithm designer to ignore many of the details of the computer on which the algorithm will ultimately be executed, but captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations is more complicated than modeling sequential computations, because in practice parallel computers tend to vary more in organization than do sequential computers. As a consequence, a large portion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the *best* model is, or about how practical various models are. Although there has been no consensus on the right model, this research has yielded a better understanding of the relationship between the models. Any discussion of parallel algorithms requires some understanding of the various models and the relationship among them.

We will now study a formulation of a parallel model based on the following example:

B

Example 4.4: Compute the sum of an integer array.

An integer array $a = \left(0, 1, 2, \cdots, n-1\right)$ is considered and the operation $\sum_{i=0}^{n-1} a[i] = S$ shall be executed. A single processor has to use a sequential, i.e. linear, approach by looping through the entire array. A parallel processor can speed up the process by computing multiple values at the same time, increasing the performance in the optimal case linearly in the number of units of parallelism. How can we map the aforementioned serialized algorithmic approach elegantly to a parallel processor model?

We now look at the abstraction of the work and depth model to find a suitable mapping for this problem. As a first indication we can use Amdahl's law to identify the speed-up factor for an algorithm when it is ported from a serial to an n-fold parallel computing machine.

D

Definition 4.6: Amdahl's Law (Amdahl [1967])

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. We now identify the speed-up by parallelization for $n \in \mathbb{N}$ parallel processors. Given $R \in [0,1]$ as fraction of the algorithm that cannot be parallelized. Then we obtained the time $T(n)$ to finish the algorithm on n processors by $T(n) = T(1)(R + \frac{1}{n}(1-R))$. The speed-up factor is determined by $S(n) = \frac{T(1)}{T(n)} = \frac{1}{R+\frac{1}{n}(1-R)}$.

Obviously, Amdahl' Law gives an indication how large the speedup can be but not a strategy to finally achieve it. This more complex consideration is taken into account by the following model.

4.3.1 Work-Depth Model

In this section we present a class of models called work-depth models to categorize and analyze algorithms. For this section we follow closely the description and notation of (Blelloch and Maggs [2004]), please see this reference for further information. In a work-depth model, the cost of an algorithm is determined by examining the total number of operations, and the dependencies among those operations. An algorithm's work W is the total number of operations that it performs; its depth D is the longest chain of dependencies among its operations. We call the ratio $P = W/D$ the parallelism of the algorithm.

The work-depth models are more abstract than the multiprocessor models. As we shall see however, algorithms that are efficient in work-depth models can often be translated to algorithms that are efficient in the multiprocessor models, and from there to real parallel computers. The advantage of using a work-depth model is that there are no machine-dependent details to complicate the design and analysis of algorithms. Here we consider three classes of work-depth models: circuit models, vector machine models, and language-based models. The most abstract work-depth model is the circuit model. A circuit consists of nodes and directed arcs. A node represents a basic operation, such as adding two values. Each input value for an operation arrives at the corresponding node via an incoming arc. The result of the operation is then carried out of the node via one or more outgoing arcs. These outgoing arcs may provide inputs to other nodes. The number of incoming arcs to a node is referred to as the fan-in of the node and the number of outgoing arcs is referred to as the fan-out. There are two special classes of arcs. A set of input arcs provide input values to the circuit as a whole. These arcs do not originate at nodes. The output arcs return the final output values produced by the circuit. These arcs do not terminate at nodes. By definition, a circuit is not permitted to contain a directed cycle. In this model, an algorithm is modeled as a family of directed acyclic circuits. There is a circuit for each possible size of the input.

Figure 4.5 shows a circuit for adding 8 numbers. In this figure all arcs are directed towards the bottom. The input arcs are at the top of the figure. Each [+] node adds the two values that arrive on its two incoming arcs, and places the result on its outgoing arc. The sum of all of the inputs to the circuit is returned on the single output arc at the bottom.

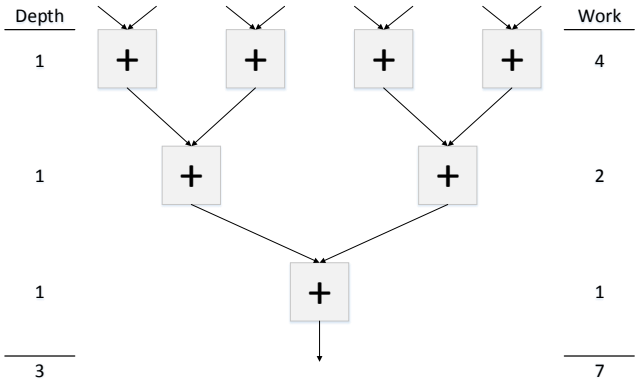


Figure 4.5: Adding 8 numbers according to the work-depth model.

The work and depth of a circuit are measured as follows. The work is the total number of nodes. The work in Figure 4.5, for example, is 7 which is also called the size of the circuit. The depth is the number of nodes on the longest directed path

from an input arc and an output arc. In Figure 4.5, the depth is 3. For a family of circuits, the work and depth are typically parametrized in terms of the number of inputs. For example, the circuit in Figure 4.5 can be easily generalized to add n input values for any n that is a power of two. The work and depth for this family of circuits is $W(n) = n - 1$ and $D(n) = \log_2(n)$.

In the work-depth models, the cost of an algorithm is determined by its work and by its depth. The notions of work and depth can also be defined for the multiprocessor models. The work W performed by an algorithm is equal to the number of processors multiplied by the time required for the algorithm to complete execution. The depth D is equal to the total time required to execute the algorithm.

Generally, however, the most important measure of the cost of an algorithm is the work. This can be argued as follows. The cost of a computer is roughly proportional to the number of processors in the computer. The cost for purchasing time on a computer is proportional to the cost of the computer multiplied by the amount of time used. The total cost of performing a computation, therefore, is roughly proportional to the number of processors in the computer multiplied by the amount of time, i.e., the work.

In many instances, the work (cost) required by a computation on a parallel computer may be slightly greater the the work required by the same computation on a sequential computer. If the time to completion is sufficiently improved, however, this extra work can often be justified. As we shall see there is often a tradeoff between time-to-completion and total work performed. To quantify when parallel algorithms are efficient in terms of work, we say that a parallel algorithm is work-efficient if asymptotically (as the problem size grows) it requires at most a constant factor more work than the best sequential algorithm known.

4.3.2 Algorithmic Strategies

The following algorithmic techniques are commonly used for cryptanalytic methods and particularly support splitting the computational load of computational algorithms among many processors. Note that this list of algorithmic strategies is not exhaustive, we only refer to most relevant ones.

Divide and Conquer: A divide-and-conquer algorithm splits the problem into subproblems that are easier to solve than the original problem, solves the subproblems, and merges the solutions to the subproblems to construct a solution to the original problem. The divide-and-conquer paradigm improves program modularity, and often leads to simple and efficient algorithms. It has therefore proven to be a powerful tool for sequential algorithm designers. Divide-and-conquer plays an even more prominent role in parallel algorithm design. Because the subproblems created in the first step are typically independent, they can be solved in parallel. Often the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism. Note however, that in order for divide-and-conquer to yield a highly parallel algorithm, it is often necessary to parallelize the divide step and the merge step. It is also common in parallel algorithms to divide the original problem into as many subproblems as possible, so that they can all be solved in parallel.
As an example of parallel divide-and-conquer, consider the sequential mergesort algorithm. Mergesort takes a sequence of n keys as input and returns them in sorted order. It works by splitting the keys into two sequences of $n/2$

keys, recursively sorting each sequence, and then merging the two sorted sequences of $n/2$ keys into a sorted sequence of n keys. To analyze the sequential running time of mergesort we note that two sorted sequences of $n/2$ keys can be merged in $O(n)$ time. Hence the running time can be specified by the recurrence which has the solution $T(n) = O(n \log n)$. Although not designed as a parallel algorithm, mergesort has some inherent parallelism since the two recursive calls are independent, thus allowing them to be made in parallel. The parallel calls can be expressed as as shown in Algorithm 1.

Algorithm 1: The Merge Sort Algorithm

```
if |A| = 1 then
  return A
  N ← −n
else in parallel do
  L = MergeSort(A[0...|A|/2])
  R = MergeSort(A[|A|/2+1...|A|)
end if
return Merge(L, R)
```

Recall that in our work-depth model we can analyze the depth of an `in parallel do` by taking the maximum depth of the two calls, and the work by taking the sum of the work of the two calls. We assume that the merging remains sequential so that the work and depth to merge two sorted sequences of $n/2$ keys is $O(n)$. Thus for mergesort the work and depth are given by the recurrences:

$$W(n) = 2W(n/2) + \Omega(n) \tag{4.1}$$

$$D(n) = max\{D(n/2), D(n/2)\} + \Omega(n) = D(n/2) + \Omega(n) \tag{4.2}$$

with sequential `Merge` of two sequences of $n/2$ entries given by $\Omega(n)$.
As expected, the solution for the work is $W(n) = O(n \log n)$, i.e., the same as the time for the sequential algorithm. For the depth, however, the solution is $D(n) = O(n)$, which is smaller than the work. Recall that we defined the parallelism of an algorithm as the ratio of the work to the depth. Hence, the parallelism of this algorithm is $O(\log n)$ (not very much). The problem here is that the merge step remains sequential, and is the bottleneck.

Sampling: Often, a problem can be solved by selecting a sample, solving the problem on that sample, and then using the solution for the sample to guide the solution for the original set. For example, suppose we want to sort a collection of integer keys. This can be accomplished by partitioning the keys into buckets and then sorting within each bucket. For this to work well, the buckets must represent non-overlapping intervals of integer values, and each bucket must contain approximately the same number of keys. Random sampling is used to determine the boundaries of the intervals. First each processor selects a random sample of its keys. Next all of the selected keys are sorted together. Finally these keys are used as the boundaries. Such random sampling is also used in many parallel computational geometry, graph, and string matching algorithms.

Symmetric breaking: Consider the problem of selecting a large independent set of vertices in a graph in parallel. (A set of vertices is independent if no two are neighbors.) Imagine that each vertex must decide, in parallel with all other vertices, whether to join the set or not. Hence, if one vertex chooses to join the set, then all of its neighbors must choose not to join the set. The choice is difficult to make simultaneously by each vertex if the local structure at each vertex is the same, for example if each vertex has the same number of

neighbors. As it turns out, the impasse can be resolved by using randomness to break the symmetry between the vertices

Load balancing: One way to quickly partition a large number of data items into a collection of approximately evenly sized subsets is to randomly assign each element to a subset. This technique works best when the average size of a subset is at least logarithmic in the size of the original set.

4.4 Lessons Learned

The following items have been discussed as part of this chapter:

- Asymptotic complexities based on the definition of Landau symbols Θ, O, Ω are widely used to relate the runtime and memory requirements of algorithms. The relation $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ is used in this context to obtain an asymptotically tight bound from the asymptotic upper and lower bounds.
- Algorithms are executed on specific computation platforms. Computation platforms greatly differ in their *processor datapath*, *feature and instruction set*, *memory architecture*, *parallelism capabilities* and *software package*.
- Modeling parallel computations is more complicated than modeling sequential computations, because in practice parallel computers tend to vary more in organization than sequential computers. Computational models such as the work-depth model help to analyze and map serial algorithms to parallel computing instances.

Chapter 5 Secret Parameters and Keys

In this chapter you will learn the basic concepts of security margins, cryptographic attacks and attacker models.

5.1 Cryptographic Secrets

As we have seen in Chapter 3, cryptographic parameters and keys are the essential parts of security systems that need to be kept secret at all costs. In this chapter we will now highlight strategies to generate, derive and store secret keys in order to assess the security risks that are present during these procedures. First of all it is helpful to have a look at security goals that are to be achieved using cryptographic systems. Commonly the following security goals are considered:

- Confidentiality
- Authenticity
- Integrity
- Availability
- Non-repudiation
- Anonymity

Obviously, some of the security goals have different requirements on the properties of involved secrets than others. For example, it is clear that confidentiality and authenticity need to bind their security to some secret parameters while integrity and availability are not explicitly requiring such parameters. For confidentiality, a secret string need to be supplied in order to successfully decrypt a previously encrypted message. For authentication we can use the following classification to distinguish the types of secrets involved and how they are derived:

- Knowledge (*what you know*): This class contains all forms of authentication that bases the authentication decision on knowledge that the user has. The most prominent example are, of course, passwords, but also PIN numbers, graphical passwords, security questions (as commonly used for account recovery), and more.
  Here, the most interesting questions are about the memorability of the secret and the entropy (or guessability, i.e., the security).
- Possession (*what you have*): This class contains a variety of authentication methods where the decision is based on possession of a cryptographic device (or security token) .
  On a high level, the main difference (as compared to the first class) is that the secret that is used to authenticate is not remembered by the user, but instead is stored on a small computing device. Consequently, entropy of the secret is not an issue, and interesting questions are how we can securely store the secret on this device (e.g., withstanding side-channel attacks).
- Properties (*what you are*): Biometric methods base the decision on specific characteristics of a user or a device. Classical examples include fingerprints, face recognition, iris recognition, and others, but also less natural schemes such as typing characteristics for uses, signal propagation delays in integrated circuits and others.
  Biometric schemes basically use fuzzy information collected by a more or less specific sensor, and compare this information to a stored sample for a specific user or device.

In light of the different options to generate and derive secrets, it is important to have an extensive understanding about the quality of the essential cryptographic secret. This includes (a) the quality (also known as *entropy*) of the secret data itself to prevent guessing attacks as well as (b) the difficulty to extract or clone this secret information from its storage or other device or human property (e.g., fingerprints of a device or an individual). The latter is particularly critical if the attacker has direct physical access to the device or individual or its/his environment.

5.1.1 Entropy of Secrets

In the next sections we will discuss different metrics and measurement approaches for the quality of cryptographic secrets. A quality measure for this purpose can be obtained from information theory and is widely known as *entropy*.

Shannon-Entropy

The Shannon-Entropy dates back to the works of Claude Elwood Shannon of 1948 (Shannon [2001]) and has many real-world applications in different fields. Using statistics as arguments we will consider a cryptographic secret as a stateless discrete random variable.

D

Definition 5.1: Shannon Entropy

Let consider $X = \{x_1, \ldots, x_N\}$ as the discrete secret space, $x \xleftarrow{R} X$ for drawing a new (true random) secret out of this space, and $p_i = P(x = x_i)$ for the likelihood of the single secret to be drawn. We denote $H_1(x)$ as (Shannon) entropy of the a source that is given by:

$$H_1(x) = \sum_{i=1}^{N} -p_i \cdot \log_2(p_i)$$

We will back up our intuitive perception of what entropy is by the following two examples:

B

Example 5.1: Equipartition/Uniform Distribution

The uniform distribution on k elements is written as U_k, a concrete example is the distribution over the four elements $\{a,b\}^2$ with

$$P(aa) = P(ab) = P(ba) = P(bb) = \frac{1}{4}.$$

For this distribution the Shannon-Entropy gives

$$H_1(U_4) = \left(-\frac{1}{4} \cdot \log_2 \frac{1}{4}\right) \cdot 4 = 2.$$

B

Example 5.2: Degenerate Distribution

Now we consider the distribution X_1 again over four elements, but the distribution is now focused in a single point:

$$P(aa) = 1, \qquad P(ab) = P(ba) = P(bb) = 0$$

For this distribution we get

$$H_1(X_1) = -1 \cdot \log_2(1) + 3 \cdot (-0 \cdot \log_2(0)) = 0$$

(where the convention is $0 \cdot \log_2(0) = 0$).

The following example will show us that the Shannon-Entropy does not capture security issues very well when it comes to heavily biased distributions. This is, for example, the case for secrets that are derived from user passwords.

B

Example 5.3: Strong Bias

Now we observe the distribution X_B with a strong bias for the first element:

$$P(x_1) = \frac{1}{2}, \qquad P(x_i) = 2^{-20} \quad \forall i \in \{2, \ldots 2^{19} + 1\}.$$

The Shannon-Entropy of this distribution is:

$$\begin{aligned} H_1(X_B) &= -\frac{1}{2} \cdot \log_2(\frac{1}{2}) - 2^{19} \cdot \left(\frac{1}{2^{20}} \cdot \log_2(\frac{1}{2^{20}}) \right) \\ &= \frac{1}{2} + \frac{1}{2} \cdot 20 \\ &= 10.5. \end{aligned}$$

This suggests rather high security and for some application this is correct. But in the field of passwords and password-hashing this is actually not the case since 50% of all passwords can be found *with a single* guess.

Min-Entropy

To take this into account, we introduce the Min-Entropy. Here we assume that probabilities are monotonically decreasing, i.e. $p_1 \geq p_2 \geq \ldots \geq p_N$.

D

Definition 5.2: Min-Entropy

Given the set of monotonically decreasing probabilities $p_i > p_{i+1}$ for draws, the Min-Entropy is defined as

$$H_\infty(X) = -\log_2(p_1).$$

This means the value of min-entropy only depends on the most common probability. This is, in the case of guessing in an interactive scenario, a good scheme to determine the security of a passwords, because in this scenario the attacker only has a very

small amount of guesses before an account is locked. For classical, offline guessing-attacks, this scheme is not appropriate, because the attacker has an unlimited amount of guesses.

Guessing Entropy

In this scheme the *expected* amount of guesses to get the secret are taken into account. The guessing entropy is not defined as a logarithmic function, but calculated the real amount of guesses needed. This results in much higher values compared to Shannon's entropy definition.

D

Definition 5.3: Guessing Entropy

The guessing entropy is defined as

$$G(X) = \sum_{i=1}^{N} i \cdot p_i.$$

For the sake of clarity we now look at several examples for guessing entropy.

B

Example 5.4: Uniform Distribution

We get the following entropy for the uniform distribution on k elements:

$$\begin{aligned} G(U_k) &= \sum_{i=1}^{k} i \cdot \frac{1}{k} = \frac{1}{k} \cdot \sum_{i=1}^{k} i \\ &= \frac{1}{k} \frac{k(k+1)}{2} = \frac{k+1}{2} \end{aligned}$$

B

Example 5.5: Strong Bias

For the same distribution with a strong bias X_B as defined above, we yield the following guessing entropy:

$$\begin{aligned} G(X_B) &= 1 \cdot \frac{1}{2} + 2^{-20} \sum_{i=2}^{2^{19}+1} i \\ &= \frac{1}{2} + 2^{-20} \frac{(2^{19}+1) \cdot (2^{19}+2)}{2} \\ &\approx 2^{17} \end{aligned}$$

Similar to Shannon's definition of entropy the resulting values are actually too high to be meaningful since still half of the passwords can be found with a single guess. This is not still yet reflected so that it needs to be refined with further strategies.

5.1.2 Secret Key Properties

In the previous section we regarded secrets as random variables. In practice a common secret (i.e., a secret key k) is realized as n-bit vector $k = (b_{n-1}, \cdots, b_0)$

with $b_i \in \{0,1\}$ that is processed by a cryptographic algorithm according to the binary standard computation model. According to the notion of entropy we can define the following properties for these bit strings to generate high quality secret parameters:

Unpredictability & Independence
All key bits b_i must be non-predictable. Furthermore, key bits b_i must neither be derived from other bits b_j nor from other known, external source or data. In other words key bits should be uniformly and independently randomly picked which maximizes the (Shannon) entropy of the key so that we can safely assume $p_{i,0} = Pr(b_i = 0) = p_{i,1} = Pr(b_i = 1) = \frac{1}{2}$. Given the uniform distribution we can expect an entropy for each key bit b_i as follows:

$$H(b_i) = -[p_{i,0} \cdot \log_2(p_{i,0}) + p_{i,1} \cdot \log_2(p_{i,1})] = 1$$

Complexity
Keys need to have sufficient complexity so that any attacker succeeds using a guessing attack with negligible probability only. In other words, the information density/entropy of an n-bit key k is given by $\mathscr{E}_k = \sum_{i=0}^{n-1} H(b_i)$ while $\mathscr{E}_k = n$ in the optimal case. Thus the attacker has an expected success probability of $Pr(Adv(A) = k) = \frac{1}{2^{\mathscr{E}_k}}$. Note that typical values of $\mathscr{E}_k > 80$ are required to resist cryptanalytic efforts by large companies or governments (cf. Table 3.1).

Persistence
Secret bits need to be securely stored or regenerated by a device under the strict control of the legitimate user only. This corresponds to the following requirements:

1. Secret parameters need to be stored in protected non-volatile key storage that is secure even under physical influence of an attacker.
2. Secret parameter alternatively need to be derived by a key recovery procedure that obtains the secret from a persistent and invariant device or human property.
3. Access to both key procedures (1) and (2) is only allowed for legitimate users that needs to provide a proof of knowledge/authentication/ownership

5.1.3 Secret Generation

In this section we identify the techniques to generate secret parameters that satisfy the aforementioned property and quality requirements. In the following we distinguish two general strategies to provide secrets.

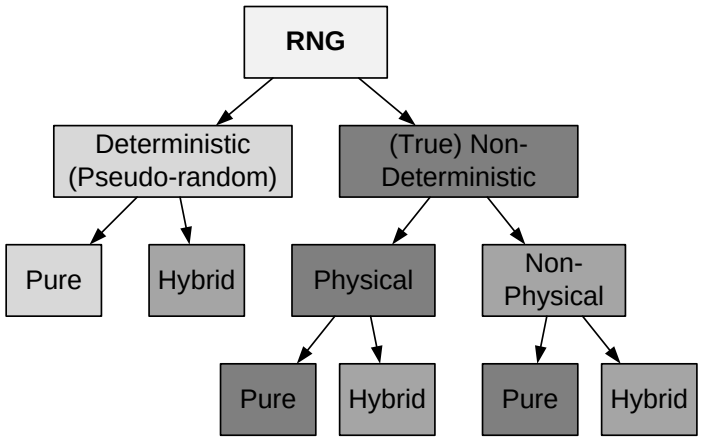
1. *Secrets obtained from Entropy Sources*. The first strategy requires the availability of a tamper-proof cryptographically strong random number generator (RNG) that provides strings of uniformly distributed random bits. An arbitrary amount of random bits are then acquired during key generation depending on the requirement of the employed cryptographic primitives. Due to the completely random nature, persistence of the secret parameters can only be achieved by embedding them into a physical storage, such as a key register, inside a cryptographic token or device.
2. *Secrets Derived from Unique Invariant Properties*. The second strategy is to obtain secret information from properties that are under strict control of a legitimate user or device. A first possibility is due to knowledge of a secret string (typically a password or passphrase) that need to be memorized by the individual. A second option is biometric data (e.g., fingerprints, iris

scans, face properties) from human beings or device-specific electrical properties of integrated circuits (e.g., *Physically Unclonable Functions* based on manufacturing-specific signal propagation delays or initial values of uninitialized memory cells). Since the latter are fuzzy and analog properties which are subject of variation due to many environmental influences (temperature, stress situation, etc.), systems based on such properties need to include facilities for error-tolerance to derive a deterministic unique secret for cryptographic use.

5.2 Secret Generation from Entropy Sources

In this section we will consider random entropy sources (also known as random number generators) that satisfy the aforementioned requirements of cryptographic applications. Figure 5.1 shows the classification of the different types of *random number generators*. Random number generators (RNGs) are generally divided into the classes of *pseudo random number generators* (PRNGs) and *true random number generators* (TRNGs). PRNGs are based on deterministic random walks and sequences to produce the uniformly distributed numbers. Obviously, if an initial value (*seed*) or intermediate value becomes known, the rest of the sequence can be reconstructed by an attacker. In contrast, TRNGs are based on physical phenomenons that are measured to obtain true randomness. Since, the generation of true random numbers is based on measuring physical phenomena, the throughput is often limited. Therefore, to increase the throughput for real-world applications, a hybrid approach is chosen combining PRNGs with TRNGs. A TRNG first generates the seed for the PRNG which provides a (deterministic) random sequence at high performance. By reseeding the PRNG from the TRNG regularly, this deterministic behaviour is mitigated of the entire construction.

Figure 5.1: Classification of Random Number Generators (RNG)



5.2.1 Pseudo Random Number Generators (PRNG)

A PRNG computes a deterministic sequence from an initial seed that shall be statistically indistinguishable from a truly random sequence. Internally the seed is passed to the initialization function in order to set up the generator. After this first step, a transformation function is iteratively called in order to update the internal state of the PRNG. On external request, random data is produced based on the current state. The internal structure of a PRNG is shown in Figure 5.2.

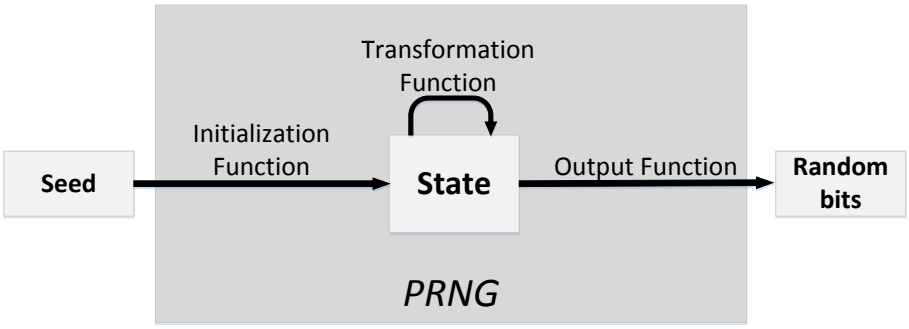


Figure 5.2: Structure of a PRNG

We now have a look how the individual functions (i.e., transformation and output functions) can be realized in practice, minimizing the implementation cost and maximizing the throughput.

Basic Pseudo Random Number Generators

Non-cryptographic applications regularly employ one of the following PRNGs. For cryptographic use, however, vanilla implementations of such generators cannot be used due to their simple predictability after a few values have become public (Paar and Pelzl [2010]).

Linear Feedback Shift Register (LFSR). For hardware-based systems, a very simple realization of a PRNG is the use of a *Linear Feedback Shift Register* (LFSR). LFSRs are very hardware-efficient since they can be implemented using only a series of flip-flops and XOR gates. The register length n of the LFSR determines the width of the state with a maximum period of $2^n - 1$. In each iteration step all state bits in this register are shifted to the right, and the gap on the left side is filled with the result of feedback function that is computed based on a selection of the state bits. The rightmost bit that drops out is the output bit. One possible realization is shown in Figure 5.3 Note that LFSR are often used as a subcomponent in cryptographic algorithms, however, due to their simple predictability care needs to be taken that an attacker cannot easily observe its inputs and outputs.

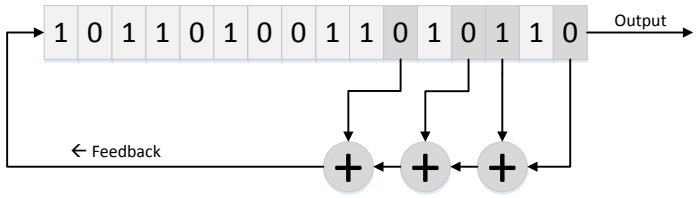


Figure 5.3: Linear Feedback Shift Register

Linear Congruence Generator (LCG). On software-based systems *Linear Congruence Generators (LCGs)* are widely deployed. LCGs use arithmetic operations that are typically available on most microprocessors and which can be efficiently performed on registers instead of single bits.
The current state (R_n) of the LCG is represented as an integer. The iteration step is described by the equation:

$$R_{n+1} = A \cdot R_n + B \bmod M$$

Note that M should be chosen to be a power of two or (better) equal to the length of the systems registers, to keep the performance of the LCG high. Finally, we should consider how to achieve the maximum periodicity of LCG-based RNGs. The following theorem states how we should choose parameters to yield a pseudo-random sequence of maximum length (Knuth [1997]).

S

Theorem 5.1: Maximum Periodicity of an LCG
The linear congruential generator defined by A, B, M and R_0 has period length m if and only if

1. B is relatively prime to M
2. $x = A - 1$ is multiple of p, for every prime p dividing M
3. x is a multiple of 4, if M is a multiple of 4

An example for a common LCG is the one provided with the ANSI C specification. The output of the `rand()` function is internally produced by an LCG with the following properties (BSD OS):

B

Example 5.6: ANSI C
$A = 1103515245, B = 12345, M = 2^{31}$
Note that only the bits $\{30 \cdots 16\}$ are used as output.

Finally we remark that LCGs have the same problem as LFSRs as they are not unpredictable in case that some output becomes known to an attacker.

Cryptographically Secure Random Number Generators

Cryptographic primitives are designed in a way that – without knowledge of the key – the output is unpredictable for an attacker given an input. More precisely, a common assumption on a secure encryption system is that its output is not distinguishable for an attacker from a random string. This property is certainly useful when designing the transformation and output function for cryptographically strong RNGs. As straightforward concept, we can use symmetric cryptography to instantiate pseudo random number generators. For this, we need to run block or stream ciphers in a recursive and stateful encryption mode. Obviously, this is directly given for stream ciphers (also called key stream generators) which are exactly designed for this purpose. However, in many (hardware) applications only block ciphers and hash functions are given as cryptography building blocks. Hence, to avoid instantiating another primitive at extra costs (namely a stream cipher) we are interested in finding alternative solutions based on available block ciphers or hash functions.

ANSI X9.17 PRNG. The ANSI X9.17 PRNG (Menezes et al. [1996]) was designed as mechanism to generate DES keys and initialization values, using triple-DES as a primitive. It is remarked that any other block cipher (E_k) can be used in the same way, too. The procedure to generate random strings based on the existence of solely a block cipher is shown in Algorithm 2.

DSA PRNG. The Digital Signature Standard specification (National Institute for Standards and Technology (NIST) [2013]) also describes a fairly simple PRNG based on SHA-1, SHA-2 (or any other hash function) which was intended for generating pseudorandom parameters for the DSA signature algorithm.

Algorithm 2: ANSI X9.17 PRNG

$T_i = E_k(currenttimestamp)$
$output[i] = E_k(T_i seed[i])$
$seed[i+1] = E_k(T_i output[i])$
return $output[i]$

Since this generator appears to come with an NSA stamp of approval, it has been used and proposed for applications quite different than those for which it was originally designed. The arithmetic of this PRNG is designed for $\bmod 2^N$, where $160 \leq N \leq 512$. The PRNG accepts an optional input W_i which could be used to state diversification. This is assumed as a zero value if not supplied. The full PRNG in pseudo code notation is shown by Algorithm 3.

Algorithm 3: DSA PRNG

$output[i] = hash(W_i + X_i \bmod 2^N)$
$X_{i+1} = X_i + output[i] + 1 \bmod 2^N$
return $output[i]$

5.2.2 Attacks on Random Number Generators

As with any other system, RNGs are also prone to attacks. The aim of these attacks is to manipulate them in such a way that the binary string produced is no longer random and maybe even predictable. When attacking RNGs we need to differentiate between attacks on TRNGs and PRNGs.

Attacks on TRNGs

The most common way to attack a TRNG is to attack the physical implementation, i.e. tampering with the hardware. One possible approach could be to increase or reduce the drive voltage above or below the allowed levels in the specification. This can also be similarly done with running the device under temperature conditions that are outside the specification. Other environmental influences (e.g., radiation, photonic emissions) can be also used to inject errors during the sampling process of the random number generator. A more active attack would be to plant a hardware trojan into the device that is activated during random number generation (Becker et al. [2013]). These errors can easily lead to statistically and abnormal output of the TRNG that finally leads to cryptographically weak keys.

Attacks on PRNGs

There are many different ways to tamper with PRNGs by running digital attacks. Following the classification methodology from (Kelsey et al. [1998]), we introduce three general attack strategies on common cryptographically secure PRNGs.

Cryptanalyzing Primitives. When an attacker is directly able to distinguish between PRNG outputs and random outputs, this is a direct cryptanalytic attack. This kind of attack is applicable to most, but not all, uses of PRNGs. For example, a PRNG used only to generate triple-DES keys may never be vulnerable to this kind of attack, since the PRNG outputs are never directly seen.

Manipulation of Inputs. An input attack occurs when an attacker is able to use knowledge or control of the PRNG inputs to cryptanalyze the PRNG, i.e., to distinguish between PRNG output and random values. Input attacks may be further divided into known-input, replayed-input, and chosen-input attacks. Chosen input attacks may be practical against smartcards and other tamper-resistant tokens under a physical/cryptanalytic attack; they may also be practical for applications that feed incoming messages, user-selected passwords, network statistics, etc., into their PRNG as entropy samples. Replayed-input attacks are likely to be practical in the same situations, but require slightly less control or sophistication on the part of the attacker. Known-input attacks may be practical in any situation in which some of the PRNG inputs, intended by the system designer to be hard to predict, turn out to be easily predicted in some special cases. An obvious example of this is an application which uses hard-drive latency for some of its PRNG inputs, but is being run using a network drive whose timings are observable to the attacker.

Compromising the State. A state compromise extension attack attempts to extend the advantages of a previously-successful effort that has recovered S as far as possible. Suppose that, for whatever reason (e.g., a temporary penetration of computer security, an inadvertent leak, a cryptanalytic success, etc.) the adversary manages to learn the internal state, S, at some point in time. A state compromise extension attack succeeds when the attacker is able to recover unknown PRNG outputs (or distinguish those PRNG outputs from random values) from before S was compromised, or recover outputs from after the PRNG has collected a sequence of inputs which the attacker cannot guess. State compromise extension attacks are most likely to work when a PRNG is started in an insecure (guessable) state due to insufficient starting entropy. They can also work when S has been compromised by any of the attacks in this list, or by any other method. In practice, it is prudent to assume that occasional compromises of the state S may happen; to preserve the robustness of the system, PRNGs should resist state compromise extension attacks as thoroughly as possible.

1. *Backtracking Attacks.* A backtracking attack uses the compromise of the PRNG state S at time t to learn previous PRNG outputs.
2. *Permanent Compromise Attacks.* A permanent compromise attack occurs if, once an attacker compromises S at time t, all future and past S values are vulnerable to attack.
3. *Iterative Guessing Attacks.* An iterative guessing attack uses knowledge of S at time t, and the intervening PRNG outputs, to learn S at time $t+\varepsilon$, when the inputs collected during this span of time are guessable (but not known) by the attacker.
4. *Meet-in-the-Middle Attacks.* A meet in the middle attack is essentially a combination of an iterative guessing attack with a backtracking attack. Knowledge of S at times t and $t+2\varepsilon$ allow the attacker to recover S at time $t+\varepsilon$.

In the following we will present a few attacks on the PRNG implementations presented above (Kelsey et al. [1998]).

Attacks on the ANSI X9.17 PRNG

This PRNG is prone to an attack if T can be fixed for a time (e.g., by manipulating the system clock as an input-manipulation attack). The X9.17 PRNG has a proven weakness (assuming a 64-bit block size) with respect to replayed-input attacks. An

attacker who can force the T values to freeze can distinguish the PRNG's outputs from random outputs after seeing about 2^{32} 64-bit outputs. In a sequence of random 64-bit numbers, we would expect to see a collision after about 2^{32} outputs. However, with T frozen, we expect a collision from X9.17 to require about 2^{63} outputs. This is a mostly academic weakness, but it may be relevant in some applications. Another reported attack is targeting the state of the PRNG. Unfortunately, the X9.17 PRNG does not properly recover from such a state compromise. That is, an attacker who compromises the X9.17 triple-DES key, K, can compromise the whole internal state of the PRNG from then on without much additional effort. There are two flaws in the ANSI X9.17 PRNG that become apparent only when the PRNG is analyzed with respect to state compromise extension attacks.

- Only 64 bits of the PRNG's state, $seed[i]$, can ever be affected by the PRNG inputs. This means that once an attacker has compromised K, the PRNG can never fully recover, even after processing a sequence of inputs the attacker could never guess.
- The $seed[i+1]$ value is a function of the previous output, the previous T_i, and K. To an attacker who knows K from a previous state compromise, and knows the basic properties of the timestamp used to derive T_i, $seed[i+1]$ is simply not very hard to guess.

Attack on the DSA PRNG

A Input-Based Attack on the DSA PRNG works in the following way. Consider an attacker controlling the input W. If these inputs are sent directly in, there is a straightforward way to force the PRNG to repeat the same output forever. This has a direct relevance if this PRNG is being used in a system in which the attacker may control some of the entropy samples sent into the PRNG. To force the PRNG to repeat, the attacker forms

$$W_i = W_{i-1} - output[i-1] - 1 \bmod 2^N$$

This forces the seed value to repeat, which forces the output values to repeat. The RNG will be stuck with the seed value. Note, however, that this attack fails quickly when the user hashes his entropy samples before sending them into the PRNG. In practice, this is the natural way to process the inputs, and so we suspect that few systems are vulnerable to this attack.

5.2.3 Hardening Random Number Generators

In order to protect the random number generator against such or similar attacks (or at least make attacks harder) the following techniques should be implemented. Some are specific to TRNGs, some are specific for PRNG and some of them are also applicable to both.

The following countermeasures are specific to TRNGs:

- Use online tests as a operational check. In case an attacker operates the device out of its specification, online tests (e.g., statistical tests on the entropy quality of the output or sensors) should be present in the device to monitor if the RNG is still fully functional.
- *Never expose the RNG output directly to the attacker.* If an attacker is successful in reducing the amount of entropy produced by a RNG, a subsequent conditioning function (e.g., a hash function) can still provide somewhat secure

cryptographic materials since the attacker cannot directly influence the RNG output anymore.

The following countermeasures are specific to PRNGs:

- *Use a hash function to protect vulnerable PRNG outputs.* If a PRNG is suspected to be vulnerable to direct cryptanalytic attack, then outputs from the PRNG should be preprocessed with a cryptographic hash function. Note that not all possible vulnerable PRNGs will be secure even after hashing their outputs, so this doesn't guarantee security, but it makes security much more likely.
- *Hash PRNG inputs with a counter or timestamp before use.* To prevent most chosen-input attacks, the inputs should be hashed with a timestamp or counter, before being sent into the PRNG. If this is too expensive to be done every time an input is processed, the system designer may want to only hash inputs that could conceivably be under an attacker's control.
- *Frequently generate a new starting PRNG state.* For PRNGs like ANSI X9.17, which leave a large part of their state unchangeable once initialized, a whole new PRNG state should occasionally be generated from the current PRNG. This will ensure that any PRNG can fully reseed itself, given enough time and input entropy.
- *Pay special attention to PRNG starting points and seed files.* The best way to resist all the state-compromise extension attacks is simply never to have the PRNG's state compromised. While it's not possible to guarantee this, system designers should spend a lot of effort on starting their PRNG from an unguessable point, handling PRNG seed files intelligently, etc.

5.2.4 Secure Storage for Cryptographic Keys and Seeds

To achieve persistence of cryptographic keys, designers need to include non-volatile storage on their cryptographic devices. This is commonly done in one (or more) of the following ways:

- *Read-Only Memory (ROM).* Although this is not a recommended strategy to implant system-wide cryptographic keys statically in the ROM of multiple cryptographic devices, this is still often done in practice (e.g., in particular to realize a *backup* or *system-vendor* key). But obviously, if this key is extracted from one device, the system security of all other devices in the field is compromised, too.
- *Electronic Fuses*. Another technique that allows the distribution of personalized keys is the inclusion of electronic fuses inside cryptographic devices. These fuses are blown during key installation in a trusted environment. Unfortunately, electronic fuses and their derivatives are large components when it comes to implementation. Even worse, "blowing" a fuse results in clearly observable physically damage of the corresponding electronic element. Hence, with appropriate microscopy it is rather straightforward to identify the secret key by optical inspection after the die has been depackaged.
- *Battery-backed RAM*. SRAM-based registers are volatile which means that they lose their contents after power down. Since SRAM registers are simple to realize and easy to obfuscate in large chips, some cryptographic devices have special battery-backed supplies to some SRAM-based key registers so that they can keep their contents even after the device is shut down. Obviously, the lifetime of the cryptographic keys are directly tied to the lifetime of the battery attached to the device.

- *Flash/EEPROM.* The most common way for non-volatile electronic storage is the use of flash or EEPROM technology in cryptographic devices. Although this has been regarded as the most secure technology for storing and hiding secret keys inside security-critical devices, there are some reports how to extract secret information from internal flash memory (known as flash bumping attacks) (Skorobogatov [2010]).

5.3 Secrets Derivation from Characteristics

If secret parameters cannot be generated from entropy sources and securely stored in protected registers, we can also use private and unique properties from human beings or electronic device to derive secrets. However, this process based on a secret derivation function needs to satisfy several requirements:

1. *One-wayness:* In case that the output of the secret derivation is leaked, it shall not be possible to reconstruct the original property (e.g., the used password or iris structure).
2. *Multi-factor process:* Some properties for secret derivation cannot be preserved in a fully protected environment. Therefore, multiple properties or property instances are taken into account for secret derivation (e.g., for using fingerprints combined with blood-circulation test).
3. *Deterministic output:* The process of secret derivation based on physical properties often comes with some fuzziness during the measurements involved. Fault-tolerant post-processing steps need to be included in the derivation procedure to account for this.
4. *Simple and efficient to use for legitimate users:* If external properties are involved, users will only accept this as source for secret derivation in real-world applications, if the additional burden and duration of the derivation process is reasonable. For example, remembering and typing a password is usually no problem for users while taking blood samples for DNA-based authentication will not be accepted.
5. *Hard to guess or recover for attackers.* The properties involved for secret derivation must be too complex and hard/impossible to acquire for an attacker to counter guessing or other recover attacks.

5.3.1 Invariant Properties for Secret Derivation

We generally distinguish between three classes of invariant properties that are used for deriving cryptographic secrets.

1. *Memorized passwords:* A simple and flexible way for secret derivation is the use of memorized secret passwords that are kept invariant for a period of time. Passwords shall be changed from time to time to enhance security. At the same time passwords are regularly forgotten so that human beings tend to have a "backup" of their password or a recovery procedure that is usually much easier to access for an attacker than a human brain. We will have a look how well typical user passwords resists common attacks.
2. *Biometry of humans:* This type of biometry uses measurable properties of a human being as recognition attributes. Common properties used for biometry are fingerprints, face-recognition, iris scans, voice recognition, the physical signature or dynamics of keystrokes when a user types on a keyboard. Generally speaking, biometric features have the strong disadvantage of a strong variance over time. Biometric characterization usually considers the quality of such properties considering:

- Uniqueness to determine how good different persons can be distinguished
- Consistency measuring that characteristics are retained over time
- Measurement efficiency and usability
- Universality to make sure that this property is globally present

For real-world applications biometry of humans are nearly exclusively used for person authentication and identification but less for generic secret key derivation. Therefore we will not delve into details in this lecture but refer to a introductory reading materials on biometry (Bolle et al. [2003]).

3. *Biometry of devices:* Similarly to biometry on humans, device properties can be measured and used for device identification or secret key generation. This concept is commonly realized by the means of so-called Physical Unclonable Functions (PUF) that are implemented on an electronic device. Typical "biometric" properties for devices are signal propagation delays, initial SRAM contents or metastability of storage cells. PUFs are often used for secret key derivation, therefore we will highlight their specific implementation later in this chapter.

5.3.2 Secret Derivation from Passwords

When using passwords to derive secrets, the strength of the password directly determines the entropy of the secret key. This can be determined from three properties:

- Length l of the password p in digits
- Size $s = |\Sigma|$ of alphabet Σ
- Distribution (if known) of each digit $p_i \in \Sigma$

B

Example 5.7: Entropy of Passwords

Let $\Sigma = \{a-z, A-Z, 0-9, +33 \text{ special characters}\}$ with $s = |\Sigma| = 95$

1. Assume a randomly picked password obtained from a uniform distribution on each element: $Pr(p_i) = \frac{1}{s} \; \forall \; p_i \in \Sigma$. How many characters are required to provide full entropy for an AES-128 key?
   $s^l = s^{H(x)}$
   $\rightarrow 95^l = 2^{128}$
   $\rightarrow l = \log_{95}(2^{128}) > 19$
2. Now assume every second digit has a constant value $p_{2i} = c$ (e.g., a hyphen):
   $\rightarrow H(p_{2i} = 0)$ Now this requires $l > 38$ to provide the same amount entropy.

Looking at the concepts of common attacks on passwords we can identify two methods to improve the security of passwords in general:

1. *Make the guessing of the passwords harder.* This is usually done by enforcing password rules. This approach is closely related to the entropy dispersion.
2. *Make the verifying of the password slower.* The verification of passwords (especially to prevent offline-guessing attacks) is to build strong password-hashes via sound and secure hash-functions.

Password Rules

A widely-used approach for password-rules are the publications of the NIST (National Institute for Standards and Technology (NIST) [2011]) that are based on Shannon's works on information theory. An administrator can choose a minimal bit-level which is necessary for a password to be accepted. Examples for such rules are:

- The entropy of the first character is taken to be 4 bits;
- The entropy of the next 7 characters are 2 bits per character; this is roughly consistent with Shannon's estimate that "when statistical effects extending over not more than 8 letters are considered the entropy is roughly 2.3 bits per character;"
- For the 9th through the 20th character the entropy is taken to be 1.5 bits per character;
- For characters 21 and above the entropy is taken to be 1 bit per character;
- A "bonus" of 6 bits of entropy is assigned for a composition rule that requires both upper case and non-alphabetic characters. This forces the use of these characters, but in many cases these characters will occur only at the beginning or the end of the password, and it reduces the total search space somewhat, so the benefit is probably modest and nearly independent of the length of the password;
- A bonus of up to 6 bits of entropy is added for an extensive dictionary check. If the attacker knows the dictionary, he can avoid testing those passwords, and will in any event, be able to guess much of the dictionary, which will, however, be the most likely selected passwords in the absence of a dictionary rule.

It is known that these rules alone cannot enforce "save passwords". If a rules enforces that at least one number must be used in the password-picking process, it is very common that the user simply appends a "1" at the end of the password. Another downside of password-rules is that generally secure (and randomly generated) passwords like "ynazwuaewfv" with ≈50 Bit (Shannon-)entropy is rated insecure whereas a the password "P@ssw0rd" is considered a strong password, although it is cracked by common password-cracker quite fast.

Password-to-Key Conversion

To be able the obtain a secret key or successful authentication using a password we usually employ cryptographic hash functions $H(x)$ as a key primitive. The following common concepts are applied for secret key derivation.

Password only: Let $H(x)$ be a selection or cryptographic hash function. Compute $H(p) = h$ and store/use h only. **Problem:** For large database files with many entries, an efficient attack is possible which mainly base on simple comparisons $h = H(p); h \stackrel{?}{=} h_1; h \stackrel{?}{=} h_2, \cdots$

B

Example 5.8: Simultaneous Attack on Multiple Passwords
Assume an adversary is in possession of u hash-values for the accounts $h_A, h_B, h_C, \ldots, h_u$ and wants to test a password-dictionary with v records $pwd_1, pwd_2, pwd_3, \ldots, pwd_v$, he performs the following:

1. $H(pwd_1) \stackrel{?}{\in} \{h_A, h_B, \ldots\}$
2. $H(pwd_2) \stackrel{?}{\in} \{h_A, h_B, \ldots\}$
3. ...

He needs to compute v hashes. The comparisons are much faster and therefore almost negligible. This leads to a very efficient attack.

Password with salt: As a remedy to the previous situation, we now introduce random salt value $s \in \mathbb{Z}_{2^t}$ with $t \leq 128$ that is hashed with the password: $h = H(p, s)$. The salt is public information and stored with h as type (h, s) in the authentication database. The length of the salt can be chosen almost arbitrary. But the longer the salt is, the safer the password is stored. The advantage of salts is that an attacker cannot pre-compute a dictionary, because he does not know the salt in advance. Even if he does, he needs significantly more storage, because each salt produces an own dictionary. In other words the salt is a prefix which creates a unique hash function for every user.

B

Example 5.9: Simultaneous Attack on Multiple Salted Passwords
Now every user password is hashed with a salt. This means that the adversary has to compute much more hash-values to recover the user passwords.

1. $h(pwd_1 \parallel s_A) \stackrel{?}{=} h_A, h(pwd_1 \parallel s_B) \stackrel{?}{=} h_B, h(pwd_1 \parallel s_C) \stackrel{?}{=} h_C, \ldots$
2. $h(pwd_2 \parallel s_A) \stackrel{?}{=} h_A, h(pwd_2 \parallel s_B) \stackrel{?}{=} h_B, h(pwd_2 \parallel s_C) \stackrel{?}{=} h_C, \ldots$
3. ...

The attacker needs to compute $u \cdot v$ hashes, if u is sufficiently large the number of hashes is drastically increased compared to the initial situation of storing plain passwords. The only advantage of the attacker is that when he gets a match he does not need to complete the dictionary, because it is very unlikely to find another database with the same salt. Therefore the attacker can stop after approximately calculating half the dictionary or 50% of the possible passwords.

Problem: Finally, we need to remark that cryptographic hash functions are still fast operations on most computing systems which still enables high throughput attacks using special hardware.

Password with secret salt (pepper): Secret salts are similar to the concept of salts introduced before with the distinction that the value s_X is not stored in the database. This secret salt is also denoted as *pepper*. The length of the secret salt cannot be chosen as freely as the length of the normal salt, because the length decides how much the computation is slowed down. The factor is: $2^t/2$. The slow-down is applied to the attacker and to the legitimate user. Typically the random secret salt $s \in \mathbb{Z}_{2^r}$ with $r \leq 16$ is hashed with the password. Since the secret salt is not stored in database a test block must be provided to verify in which case the correct secret salt is found (cf. TrueCrypt).

Iterative password hashing: A way to further improve the introduced countermeasures against password-thefts is making the hash function H slow. This

slows an attacker even more, because he needs more time to compute the hashes. On the other hand the hash function can not be too slow. This would impair the user when he wants to log-in and needs to wait for several seconds/minutes. A good trade-off must be achieved. In the following we will show several ways of how to realize such a slowed down hash-function. Assume that a password x is processed y times by derivation function $h = H^y(x)$ with y being a variable parameter adaptively chosen according to the complexity of $H(x)$ and available processing power. Typical values for this iteration parameter are $1024 \leq y \leq 4096$.

Password Systems in Practice

We now introduce some historical derivation techniques that were and are used in many common password-based security systems. Although some of them are obsolete today, their concept still underlines the importance of some critical aspects in secret derivation procedures that need to be considered to maintain a secure system.

DES-crypt The Unix function `crypt()` is based on the Data Encryption Standard (DES) and is obsolete today but still an interesting historical example. It uses 25 iterations of a slightly modified DES algorithm. The modification causes that the computation time is much higher than the standardized DES algorithm. This comes with two advantages. First, the attacker needs longer to set-up a dictionary and recover the passwords. The second advantage is that the attacker can not longer use dedicated hardware which is specialized on cracking DES encrypted passwords. This hardware is very popular, because DES is (or was) widely used and established.
In detail the modification uses a salt of a fixed length and uses this value in the expansion-permutation-function and puts out one of $2^{12} = 4096$ different values.

1. The password has a length of 8 characters. If it is too long it will be shortened, if it is too short it is padded with zeros.
2. The resulting string has a length of 56 bits (encoded as ASCII characters) and is regarded as key to encrypt a fixed string with 25 iterations of the modified DES function.

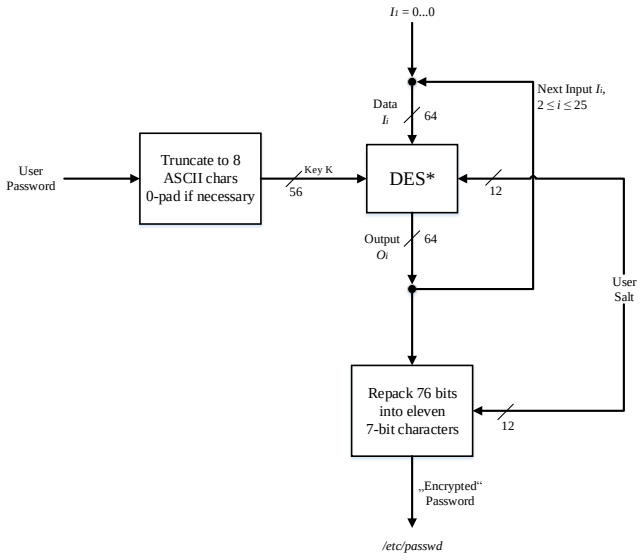
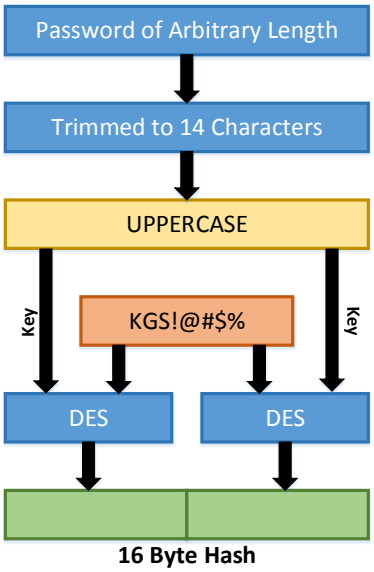


Figure 5.4: DES-crypt. [Quelle: Menezes et al. Kapitel 10]

NT LAN Manager (NTLM) The NT LAN Manager (NTLM) was used in the earlier versions of the Windows operating system (Windows NT) and is considered insecure nowadays.

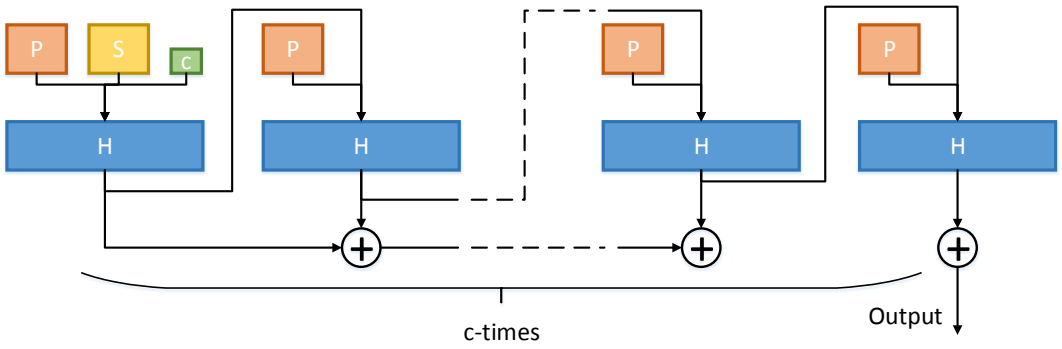
1. The password is shortened down to 14 characters (or padded with zeros if it is shorter). Then every character is converted into uppercase.
2. The password is split into to halves (each 7 characters). Each half is the key for a DES algorithm and encrypts a fixed string.

Figure 5.5: NT LAN Manager Hash



PBKDF2 The password-bases key derivation function 2 (PBKDF2) is a standardized key-derivation function (PKCS #5 v2.0, RFC 2898) and is widely used today to store passwords securely (Kaliski [2000]). Examples for their use is TrueCrypt, WPA or the encryption of the OpenDocument format (TrueCrypt, Ope [2011]). The central component of PBKDF2 is an integrated HMAC construction that is iteratively called c times. The number of iterations can be chosen freely as well as a wide range of different hash-functions can be used.

Figure 5.6: PBKDF2



Attacking Passwords

In this section we want to categorize and formalize a common strategy to attack passwords and their respective derivation schemes. Consider the following notation that we will use for later attacks on passwords.

- Σ Alphabet (terminal symbols) for the password (e.g., $\Sigma = \{a-z, 0-9\}$)
- P Finite set of production rules. Rules can be associated with with weights and probabilities if distributions are known.
- D Finite set of constant words over Σ to simplify P: $D = \{w : w = \sum_{i=2}^{n} a_i \; ; \; a_i \in \sum \; ; \; 2 \leq n \leq l\}$
- l Fixed or maximum length of possible passwords.

Based on this notion we introduce two main components of the attack: (a) the password iteration unit (PIU) and (b) the password validation unit (PVU). Both components are shown in Figure 5.3.2 and can be configured to run a wide range of different attack schemes.

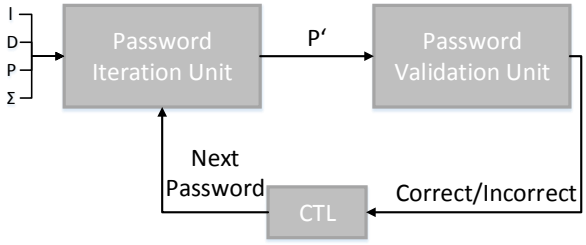


Figure 5.7: Attack Scheme on Passwords

Remarks:

1. The PIU and PVU each can be realized on separate platforms.
2. PIU output is compiled as a large list of potential passwords that can be transferred in one block to the PVU by bulk data transfer. This simplifies and accelerates direct access to password candidates at the cost of high(er) temporary storage requirements.
3. If P and D are large they can be distributed over several parallel computation platforms.
4. Performance of the password attack is determined by the latency of the password verification cycle (PIU→PVU).

Password Iteration Unit (PIU)

There are multiple strategies to effectively search through the password search space, depending on the type and structure of the passwords. Most common strategies are covered in this section.

1. *Exhaustive Search:* Production rules test all combination of p with (up to) l chars. The resulting complexity for this search is $|\Sigma|^l$.
2. *Regular Expressions:* Production rules build all possible combinations defined by a regular expression that was built according to some apriori knowledge of the attacker on the password. The resulting complexity obviously depends on the range of the regular expression $|L|$ with $L = \{w : Regex(i) = w\}$.

3. *Dictionaries:* Often based on social information that is compiled into a dictionary D, we are left with a trivial production rule set P that just iterates over D. The time complexity to complete this attack is $|D|$.
4. *Mangling Rules:* The use of mangling rules is based on a hybrid of regular expressions and a dictionary attack.
5. *Markov Chains:* Production rules follow probability for a symbol, exceeding a threshold of δ. This technique can be additionally combined with subsequent filters to reduce the number of returned password candidates.

We now briefly introduce the different concepts:

Regular Expressions/Grammars

Regular grammars describe a regular language based on the formal grammar $(N, \sum, P, S)$ with non-terminals N, terminals T, a production or rule set P and start symbol S:

- General (right-regular) form for $P : X \to aX, X \to a; X \to \varepsilon$ where $X \in N$, $a \in \sum$ and ε the empty String
- Follows standard concept to parse information (right $\to$ left)

Regular expressions (also called *RegEx*) are a simplified way to describe words in regular languages (e.g., as search pattern). Typical elements of regular expressions are:

- Metacharacters with its special coding instruction
- Regular characters with literal meaning (terminals)

Note that different notations of RegEx are possible, e.g., `[+-]?[0-9]+(,[0-9]+)?`. For this lecture, we follows the syntax used by the `RegExLib` (RegExLib) as shown in the Appendix. In the context of password search, we now introduce two possible example notations:

B

Example 5.10: Search Expression # 1: `^[a-zA-Z]\w{3,14}$`

The password's first character must be a letter, it must contain at least 4 characters and no more than 15 characters and no characters other than letters, numbers and the underscore maybe used. Possible patterns are: `abcd, aBc45DSD_sdf, password`

B

Example 5.11: Search Expression # 2: `^(?=.*\d).{4,8}$`

The expression specifies a string (or password) that is between 4 and 8 digits long, and includes at least one numeric digit. For example: `1234, asd1234, asp123`

Dictionaries

A dictionary is a precomputed list of potential passwords, i.e. a large set of independent entries. We can assume that each key in the set is unique, since there is no advantage in storing equal entries multiple times. Dictionaries are typically used when the output is hard to derive algorithmically. Especially in restricted devices

(restricted in regards to computational power), dictionaries can yield great speed improvements. In the case of password cracking dictionaries or password lists, are used to minimize the latency between the PIU and the PVU. Dictionaries can also be used to enable the transfer between different platforms acting as the PIU/PVU. The first step is to acquire or to generate a dictionary before one can start to attack a password with a dictionary attack. When cracking a large bulk of passwords, or a single password with not much background information about the owner/creator of the password, it is advisable to use existing word lists. Sources are general word lists, encyclopedias or dictionaries (e.g., database from wikipedia). Another option to generate case-specific dictionaries based on following sources:

- Social information of the target person whose password is attacked
- Poems and song lyrics in the language(s) spoken by the target
- Bookmarks or popular websites which have been accessed by the target
- Keyboard walks (i.e., qwert, 1234, 1q2w3e4r and so on) which are common for human behavior

Given such information a general approach to generate such work list is to (a) create individual entries (e.g., replace spaced by newlines), then (b) remove all duplicates and (c) sort all entries according to their optimal processing order (e.g., by number of characters).

If this information is not available or insufficient, an alternative source for dictionary-based attacks are leaked password lists. These passwords lists got public due to security breaches of several companies. The revealed password lists contain the passwords of thousands of real users in plain text. These lists can also be used to train and validate password generators. Probably the best-known list is called **RockYou** and contains 14.341.086 real passwords[1]. A listing with several leaked password lists is given in Table 5.1.

Database Name	**Number of Unique Passwords**
Rockyou	14.341.086
zappos	2.150.832
Singles	24.870
Phpbb	259424
Stratfor	860.160
Gawker	743.864
Tianya	40.000.000
Sony	77.000.000
Likedin	6.458.020
alybaa	1.384

Table 5.1: Leaked Password Lists from Security Breaches

These lists rarely eliminate all duplicates in the list. Table 5.2 shows the number of occurrences of the top 10 passwords in the RockYou list. Note that bold entries could be similarly successfully attacked with a simple regular expression `^[0-9]{5,9}$`.

Mangling Rules

Mangling rules combine properties of dictionaries and regular expression. They are used to modify individual dictionary entries, for example chop-entries or add prefix/suffix to get new passwords. It is also possible to combine several entries

[1] Note, however, that the quality of passwords used for a service might strongly differ to the importance of the protected information. It is obvious that RockYou passwords are obviously not as carefully picked by individuals compared to their choices on the passwords to protect their bank accounts.

Table 5.2: Top 10 Passwords from RockYou Dictionary

Rank	Num of Occurrences	Password
1	**290.729**	**123456**
2	**79.076**	**12345**
3	**76.789**	**123456789**
4	59.462	password
5	49.952	iloveyou
6	33.291	princess
7	**21.725**	**1234567**
8	20.901	rockyou
9	**20.553**	**12345678**
10	16.648	abc123

from dictionaries. They can be implemented by applying functions to dictionary entries (e.g., capitalize, first letter only, etc.). This section will be extended in the next release of the lecture materials.

Tools for Password Attacks

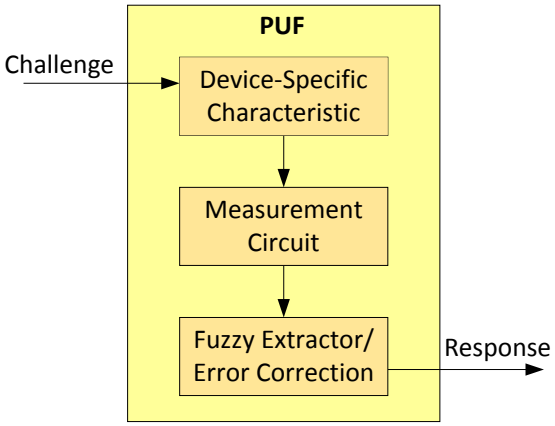
This section will become available in the next release of the lecture materials.

5.3.3 Key Derivation from Physically Unclonable Functions

A PUF (*Physically Unclonable Function*) is a function that cannot be duplicated with physical methods[2]. The function is evaluated by measuring physical properties on integrated circuits and thus not deterministic.

The fundamental principle of a PUF can be compared to a challenge and response protocol. In this protocol the function receives a challenge c and it produces a unique response r.

Figure 5.8: PUF Concept



Internally, a PUF has three main components as shown in Figure 5.8. At first the challenge is provided to the measurement circuit to query the device unique properties at a specific position/angle. The returned value is affected by errors and thus needs extra post-processing (by so called fuzzy extractors based on error-correcting codes) to return a deterministic response.

[2] That does not mean that it cannot be cloned with mathematical models.

Properties of PUFs

One of the most important characteristics of PUFs is the distance of responses. A response to the same challenge should have a maximal distance on two different devices. The distance is computed as hamming-weight between two output values, we will refer with μ_{inter} to this distance between different devices. For an optimal distance this value should be $\mu_{\text{inter}} = 50\%$.

Another important property is that the same PUF needs to respond persistently in the same way. That means, the input of the exact same challenge on the same device, should result in a minimal distance in the outputs of the different measurements. This distance is called μ_{intra}. The optimal case is that $\mu_{\text{intra}} = 0\%$.

The following properties of a PUF are usually relevant for security applications:

- *Evaluation*: The function $y = \text{PUF}(x)$ must be evaluated easily.
- *Uniqueness*: $\text{PUF}(x)$ has to contain unique information about the physical identity of a system.
- *Reproducible*: $\text{PUF}()$ can be a probabilistic procedure, but $y = \text{PUF}(x)$ needs to be reproducible (except of a small error).
- *Non-Clonable*: It need to be hard to create a $\text{PUF}'(x)$ to a given $\text{PUF}(x)$, such that $\text{PUF}'(x) = \text{PUF}(x)$.
- *Non-Predictable*: It needs to be hard to predict $y_0 = \text{PUF}(x_0)$ given other values x_i, y_i.
- *One-wayness*: With a given y and $\text{PUF}()$ it need to be hard to find a x, such that $y = \text{PUF}(x)$.
- *Tamper-Evident*: A change in the physical system which also affects the $\text{PUF}()$ has to change the response.

PUF Instances

We now provide a few examples how PUFs are implemented on electronic devices.

Delay-based Arbiter PUF Delay-based PUFs exploit different signal propagation delays in logical paths that originally have been laid out in a completely identical way. To drive this propagation paths according to a challenge, they introduce series of connected multiplexers. These muxes route the input signals straight if the corresponding challenge bit is 0, and cross the inputs symmetrically if the challenge bit is 1. With n of these switches connected in series, we can feed in a n-bit challenge controlling signals of n blocks. The response consists of one bit that identifies which signal reached the end at first. Figure 5.9 shows the concept of the Arbiter PUF.

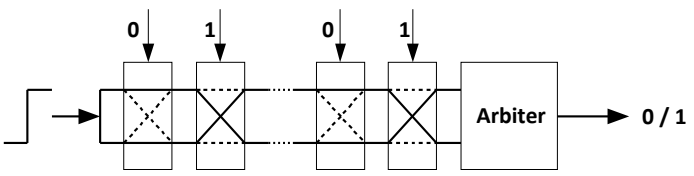


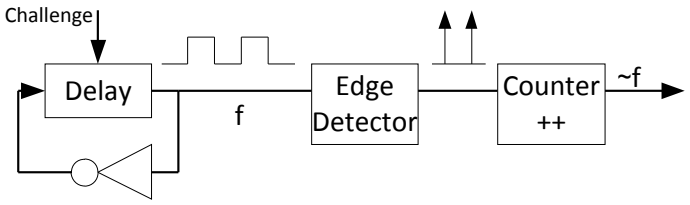
Figure 5.9: Functionality of a Arbiter-PUF

The construction was introduced in 2004 by Lim et al. And was tested for 10.000 challenge and response tuples on 37 ASICs for $n = 64$. This leads

to $\mu_{\text{inter}} = 23\%$, which is not optimal and $\mu_{\text{intra}} \approx 0,7\%$, which is close enough to 0.

Delay-based Ring Oscillator PUF Another common concept of PUFs is based on oscillators originally presented by Gassend et al. in 2002. A delay stage that is similar to the one as for the Arbiter PUF is inserted as component of a ring oscillator (RO). The challenge specifies the configuration delay element which has a direct influence on the oscillation of the RO. The number of oscillations is sampled and evaluated as PUF response. Figure 5.10 shows how this response is generated by using a counter fed by an edge detector.

Figure 5.10: Ring Oscillator PUF by Gassang et al.



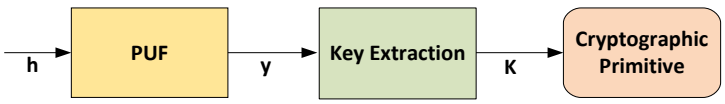
In order to compensate for other environmental influences (e.g., temperature), two oscillators are instantiated in parallel and their counter output divided by each other. The ratio of the two ROs that are both affected of environmental influences in the same way are much more robust against these distortions. Another option with smaller resource footprint is by just comparing both ROs returning a single bit which of the ROs has the higher frequency. For multiple response bits, we need provide or measure two oscillators for each of these bits.

Employment of PUFs

PUFs are employed using a two-stage protocol. In the enrollment phase, a database is created recording many challenge-response tuples for every PUF. This is typically done in a trusted environment during/after manufacturing. In the reproduction phase that is run when the device is in the field, the response is compared with the values in the database.

PUFs have several use-cases. First they can be used to identify a physical system and device. Second, it is possible to derive a secret key from the a PUF as shown in Figure 5.11. This is done by using a fixed challenge that is also called helper data h. The obtained response is then taken as a cryptographic key and is fed to subsequent cryptographic primitives.

Figure 5.11: Key Generation with a PUF



Note that PUFs are capable to (re-)store a secret in a non-digital way on an electronic device. The keys are only derived on demand so that in all other cases the keys are not physically present inside the device.

5.4 Workshop: Applied Password Guessing

Background on the Problem: You were hired at the medium-business company ShaSecure. For security and safety reasons, the company uses a centralized backup server to which the local data of every employee's computer is automatically replicated. All personal data of each employee is stored in an encrypted truecrypt drive to ensure confidentiality that employs the cipher suite based on RipeMD160 and AES-XTS. Since in the past some employees did not use strong passwords to encrypt their drive, a tool was written to create passwords for the employees. The tool generates a 25-digit string consisting of capital letters (A-Z) and numbers (0-9). The tool was designed to provide strong security based on symmetric cryptography with more 128 bits. To prevent employees from forgetting their password (which would lead to a massive bureaucratic effort and/or loss of data), a deterministic algorithm was chosen to create personalized password strings. You realize immediately that this solution might be prone to attacks and demonstrate this to the management by supplying a working attack on a truecrypt drive of an arbitrary employee (of course – to which you are actually not supposed to have access).

Details on the Password Generator: After several discussions with the system administrator, you retrieved the details of the password generation program. The first nine digits represent the company name. The next eight digits are generated using a LCG with the parameter set $m = 2^{32}$, $a = 1664525$ and $c = 1013904223$. The employee's date of birth is used as the initial parameter x_0 and the number of iterated invocations of the LCG are determined by the employee's personal number in the company. The day and month of the birthday are stored in one byte each and the year in two bytes. The values are then concatenated (`day|month|year`) where `day` corresponds to the most significant byte. The output of the LCG in hexadecimal notation is inserted into the password. The next three digits indicate the name of the department the employee. The last five digits represent of the year when the employee was hired and the security level of the employee.

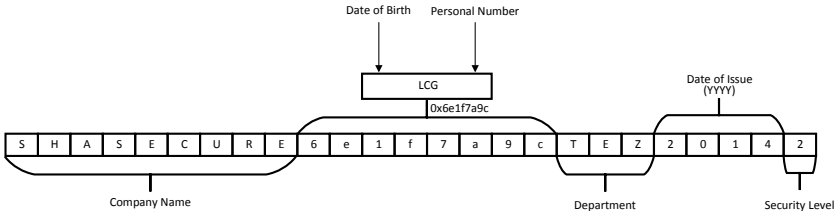
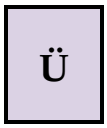


Figure 5.12: Structure of the Passwords

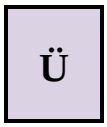
About the Company: ShaSecure was founded on 1 January 2012. Two and a half years later it employs 65 people. Each new employee is given a personal number (which is issued sequentially) and a security level (1-4). Then they are assigned to one of eight different departments (please see the list given in the Blackboard) depending on their field of expertise. The company follows good ethical standards and does not support child labor. Furthermore, employees are encouraged to retire at the age of 65.

You are now asked to crack the password of the truecrypt file container and to complete the following tasks.



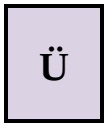
Exercise 5.1

Make appropriate assumptions about the input parameters for the password generation tool and estimate the entropy.



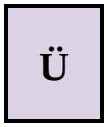
Exercise 5.2

Create a dictionary with potential passwords using the aforementioned information about the password generation program.
Example for the LCG: Date of Birth: 15.05.1986, PN: 25, $x_0 = 251987906$, output: `0xefb00391`.
BUG: There is a bug concerning the containers that are uploaded in Blackboard. The letters of the LCG output need to be lower case.



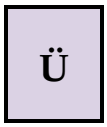
Exercise 5.3

Develop or use an appropriate tool to feed the dictionary into a password attack on truecrypt containers. If possible make use of the power of graphics cards for your attack. Run the attack which will certainly take some time (depending on the quality of your assumptions and implementation). While you are waiting, please consider to continue with tasks (f), (g) and (h) and/or start to upload some of your solutions to Blackboard.



Exercise 5.4

Check the throughput of your attack, time consumption of your attack and the platform details you are using for running the attack. This will be used to compare the efficiency of your solution afterwards.



Exercise 5.5

Use the recovered password to access the data inside the container and upload it to Blackboard as a proof of your success.

Exercise 5.6

Assuming the entropy of the password generation tool is 64 and 128 bit. Estimate the duration you need to search for the correct password based on this security level and the throughput of your system.

Ü

Exercise 5.7

Assume you are given $100,000 and your goal is to build a GPU-based truecrypt cracker. Find the current price of your GPU and estimate the throughput of a $100,000 cracker (assume 33% overhead for the base platform for each GPU). How long does it take for a cracker to find a password with the entropies given in (a) and (f). Extrapolate the results for a "sweet spot"-GPU offering best price/performance ratio (in May 2014) and estimate the improvement factor using this GPU platform on the cracking device.

Ü

Exercise 5.8

Make a proposal to harden the given password generation process against dictionary attacks.

Ü

5.5 Lessons Learned

The following aspects of secret generation and derivation have been discussed in this chapter:

- A quality measure on secrets to prevent attacks in the *entropy*.
- Different random number generators can be used to generate secrets with high entropy. Appropriate cryptographically secure pseudo-random number generators (CSPRNG) or true random number generators (TRNG) are suitable choices.
- Secret derivation from properties is possible from user passwords or device properties using Physical Unclonable Functions (PUF). While the former are troublesome with respect to password quality and powerful password attacks, the latter need expensive error-correction and cryptographic post-processing to provide deterministic and unique secret keys.

Chapter 6 Tools for Symmetric Cryptanalysis

Symmetric cryptography is a key component in many digital security systems to realize encryption and authentication. In this chapter you will learn generic ways and methods to attack (a) symmetric block ciphers and (b) cryptographic hash functions.

6.1 Symmetric Constructions

Symmetric cryptography mainly consists of stream ciphers, cryptographic hash functions, and block ciphers. In the following, we explain the latter two in more detail as they are relevant to the content of this chapter.

6.1.1 Block Ciphers

Given a plaintext P_0 and ciphertext C_0 with blocksize b encrypted under cipher S and key $k \in N$, encryption is defined as $C_0 = S_k(P_0)$. In this scenario, an attacker's task is to invert S_k without knowledge of k.

There are a few general assumptions that have to be made for secure ciphers. At first, the best possible attack is exhaustive search on key space N. Second, the produced ciphertexts possess the statistical properties of pseudo-random numbers.

Definition 6.1: Search Problem on Symmetric Keys

The search problem on a key space of size N is defined by the following two complexities:

1. Time T specifies the number of involved operations
2. Memory M specifies the words of memory required for the attack

D

The definitions of a word of memory and of an operation is specific to the attack. In the following we will analyse three attacks and their specific complexities.

Exhaustive Search [T=N, M=1]

Since there are several different scenarios for exhaustive search attacks, we assume a known-plaintext attack which is the most trivial cryptanalytic attack. The memory needed is equal to one, as only the plaintext has to be stored and compared with the output of each iteration.

Table Look-Up [T=1, M=N]

This attack requires a chosen-plaintext scenario, i.e. the plaintext is fixed and encrypted under all possible keys. The table is precomputed and time complexity to generate the table is not part of the actual attack. The attack is effective only if the plaintext is often reused for encryption under different keys.

Time-Memory Trade-Off [1<T<N, 1<M<N]

This attack combines the first two attacks and, in this way, can improve the overall attack complexity. Likewise, all restriction of the first attacks also apply to this

one. The attack is also probabilistic and is therefore likely to fail under certain circumstances.

6.1.2 Cryptographic Hash Functions

Cryptographic hash functions map inputs of large (nearly) arbitrary lengths to outputs with a fixed length. Most hash functions chop the input into blocks and words that are mixed in the hashing process.

Given the hash function $H(m)$ with the properties:

1. *Preimage Resistance:* Given a hash value h it should be hard to find any message m such that $h = H(m)$.
2. *Second Preimage Resistance:* Given a message m_1 it should be hard to find another message $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$.
3. *Collision Resistance:* It should be hard to find different messages m_1, m_2 such that $H(m_1) = H(m_2)$.

The worst-case complexities of generic attacks on hash functions with k bit output on the respective properties are

1. *Preimage Resistance:* 2^k via exhaustive search for matching preimage
2. *Second Preimage Resistance:* 2^k via exhaustive search for matching hash digest
3. *Collision Resistance:* $2^{k/2}$ via birthday collision attack with square root complexity

For example, in the case of MD5 (with $k = 128$) attacks on the properties (1) and (2) are infeasible with today's resources since 2^{128} cannot be cracked in practicable time. However, an attack with a complexity of 2^{64} on the collision resistance is potentially feasible but still very expensive.

6.2 Time-Memory Trade-Off Attacks

6.2.1 General Principle

The principle of Time-Memory Trade-Off (TMTO) attacks is based on chained encryptions that are partially stored in memory to reduce the time of cryptanalysis. In 1980, Martin Hellman introduced the time-memory trade-off attack which generates tables of chained encryptions. An encryption chain is informally denoted as chain and is generated as follows:

1. Specify a fixed plaintext P_0 and a cipher suit $S_k(x)$ for which the attack is conducted.
2. Define a step function $f(k) = R(S_k(P_0))$ where $R(x)$ is a simple (reduction) function that maps the ciphertext to the keyspace domain: $\{0,1\}^b \to N$.
3. A chain of length t is generated by t iterations of $f(k)$. The resulting chain $k_0 \to k_1 \to \cdots \to k_t$ is computed by $f^t(k)$.

6.2.2 Attack Specification

The attack is split into two phases, the precomputation phase and the attack phase. The *precomputation phase*, which can be done offline, i.e. without starting the actual

attack, generates encryptions and consists of the following steps for the classical Hellman tables:

1. Pick a suitable plaintext P_0.
2. Pick m random keys k_i from the key space as the starting points $SP_i = k_i$ with $i = 0 \dots m-1$.
3. Compute corresponding end points $EP_i = f^t(SP_i)$ as

$$\begin{aligned} SP_1 &= K_{(1,1)} \to K_{(2,1)} \to \cdots \to K_{(t,1)} = EP_1, \\ SP_2 &= K_{(1,2)} \to K_{(2,2)} \to \cdots \to K_{(t,2)} = EP_2, \\ &\vdots \\ SP_{m-1} &= K_{(1,m-1)} \to K_{(2,m-1)} \to \cdots \to K_{(t,m-1)} = EP_{m-1}. \end{aligned}$$

4. Store all tuples (SP_i, EP_i) in one table and sort after EP_i
5. Repeat steps 1-4 to create additional tables based on a modified reduction function $R'(k)$.

The *attack phase*, i.e. the online phase, is used to actually obtain the plaintext P_1 for given a ciphertext Y_1. Informally, this is done by checking if Y_1 is an element of the stored chains. The full key recovery is done by performing the following steps:

- *Find matching (SP, EP) tuple:*
  For this we need to apply the reduction function $R(Y_1)$ and check if the result matches an endpoint in the table. If it does, the key $k_{t-1,i}$ which is the second to last element of the chain is a possible candidate. In this case, the (SP, EP) tuple is marked as a matching chain and the key recovery and validation process is triggered. If $R(Y_1)$ does not match to any EP in the table, $R(Y_2) = f(R(Y_1))$ is applied and the searching process for a matching endpoint is repeated. If it matches, the key $k_{t-2,i}$ is a possible candidate. By repeating these steps, the chains are checked iteratively for the given ciphertext by just knowing the endpoints. Beside the search, the found key candidates have to be recovered and verified.
- *Key recovery and validation:*
  To recover the key from a matching chain, the chain has to be recomputed from the identified starting point SP. The potential key candidate is then checked if it decrypts Y_1 correctly. If the decryption returns the wrong result, the first step is repeated to find a new tuple.

A wrong encryption can happen due to so called *false alarms*. A false alarm occurs if chains merge. A merge appears if the reduction function R reduces the output of two or more chains to the same key k_i. Hence, the chain keys following a merge are identical. The occurrence of false alarms is given by the following example:

Given is a ciphertext Y_1 and two chains with starting points SP_1 and SP_2 and end points EP_i and EP_j. The merge occurs at $k_{1,i+1}$ and $k_{2,j+1}$ so that the following keys are identical. During the tuple search process, the chains are checked iteratively from EP to SP by assuming $R(Y_1) = k_{x,y}$ at a specific position in the chains, depending on the processed iteration. We assume that the correct key is $k_{2,j}$. Due to the merge, the tuple (SP_i, EP_i) is still marked as a valid chain and therefore k_1, i is also identified as a valid key. This phenomena is depicted in Figure 6.1.

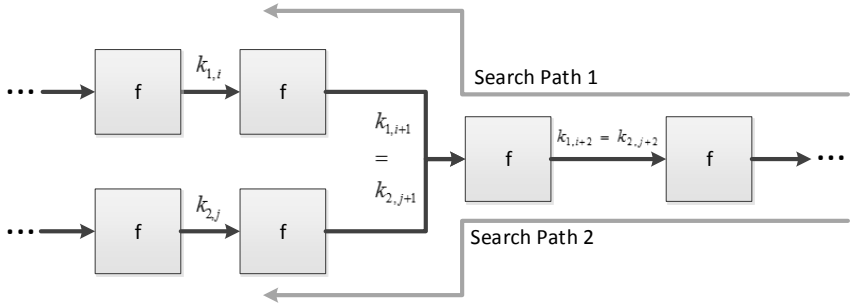


Figure 6.1: Merge of two chains.

6.2.3 Attack Parameters

As mentioned before, TMTO attacks are probabilistic attacks with a specific success probability, based on the following parameters:

- m: the number of words (tuples) in a table
- t: the length of the computation chain
- l: the number of used tables

These parameters define the coverage of the key space and influence the success rate. Determining the exact success rate is a non trivial problem. The existence of merges and cycles in a chain lowers the covered key space of the table which is hard to predict. For a success rate estimation, a lower bound formula for using one table, containing m tuples and a chain length of t is given by:

$$P_{table} \geq \frac{1}{N} \sum_{i=0}^{m} \sum_{j=0}^{t-1} (1 - \frac{i \cdot t}{N})^{j+1}$$

The overall probability of success with l tables is determined by:

$$P_{success} \geq 1 - (1 - P_{table})^{l}$$

Beside the success rate, the parameters also influence the run time of the precalculation phase T_P, the run time of the attack phase T_A, the required table look-ups TLU and the size of required memory space M.

The memory space is given by

$$M = m \cdot l \cdot m_0$$

while m_0 is the memory space, required to store one tuple.

The run time of the precalculation phase is given by

$$T_P = l \cdot m \cdot t \cdot t p_0$$

while $t p_0$ is the time required to generate one element of the chain. The precalculation phase can be parallelized and strongly depends on the used hardware.

The estimation of the attack phase run time is also a non trivial problem. The attack phase run time is strongly influenced by the occurrence of false alarms. Each false alarm requires a key reconstruction and validation step which results in

an increased run time. The occurrence of false alarms is based on the amount of merges which is like previously noted hard to predict. A worst case estimation for the run time of the attack phase neglecting false alarms is given by

$$T_A = l \cdot t \cdot ta_0$$

while ta_0 is the time required to generate one element of the chain and to search the table for the element which includes the communication delay between the processing unit and the storage element. The communication delay between processing unit and storage element stronger comes into account the more table look-ups have to be processed. Assuming one table look-up to search for an element,

$$TLU = l \cdot t$$

table look-ups are required in worst case. The run time of the attack phase can also be reduced by parallelization of the process and strongly depends on the used hardware.

In the following the Data Encryption Standard (DES) is examined to show the efficiency and success rate of this attack. The attack on DES is specified by $t = 2^{19.2}$, $m = 2^{16.7}$, $l = 2^{21}$. This results in a lower bound success rate of 80% and requires a storage space of $1.8TB$. In order to create the precomputation table the COPACABANA needs 24 days due to the slow communication interface between the FPGA cluster and the connected hard disk. The online phase requires a run time of $2^{40.2} \cdot ta_0$.

6.2.4 Extensions and Improvements

The Hellmann approach has several problems and disadvantages. For instance each computation requires a table access in the online phase to check if the computed value corresponds to a stored endpoint. Also false alarms are possible, which are created through merging multiple paths. At last, cycles can exist which reduces the key space coverage by looping entries.

One improvement on the classical Hellmann tables was introduced 1982 by Rivest; the so called *Distinguished Points*. Rivest defined alternative endpoint criteria for k_i, which are the distinguished points (DP). A possible criteria could be x leading/trailing zeros. The chain lengths are therefore no longer constant depend on the parameter x. In general the chains become longer than classical chains. However, the success probability stays the same. The known DP structure keeps the table accesses equidistant and reduces the table look-ups by roughly a factor of x, but false alarm detection requires a higher overhead due to longer chains. Another advantage of this approach is that loops can be easily detected, since DPs are expected to be found with probability $> 50\%$ after x steps. If no DP is found after $>> x$ steps, the computation has probably reached a loop. Also merged chains can be removed when it is detected that they result in the same endpoint. The precalculation phase comes with a higher computation time. This results from merged chains that are more likely to occur due to the longer chains and chains that do not end in a DP within x average steps, including cycling chains.

Another approach is the use of so called *Rainbow tables*. This approach was introduced by Oechslin in 2003. A rainbow table defines a sequence of reduction functions R_i which differs for each step in the chain: $f(k) = R_i(S_k(P_0)))$. Rainbow tables come with several advantages. At first even if chains collide, they usually do not merge, since the reduction functions are specific to each round. A chain merge only happen if the collision occurs in the same step. If a merge occurs, they

can be detected since they will result in the same endpoint, that can be filtered from a table. Loops are completely avoided due to the use of round dependent reduction functions. The rainbow table chains have a constant length. So they are very well suited for big table sizes. Roughly, t Hellman tables with a table size t can be placed in one Rainbow table to keep an equal success rate. The large rainbow tables reduce the amount of table accesses by a factor of t compared to Hellman tables. The different reduction functions for each step requires a recalculation of all steps taken backwards from the EP to the destination element. Due to the linear increase of calculation steps over time, the table access rate reduces over time. The different behavior and properties of Rainbow tables result in a different success rate estimation and run time of the attack phase.

$$
\begin{aligned}
P_{table} &\approx 1 - \prod_{i=1}^{t}(1 - \frac{m_i}{N}) \text{ where } m_1 = m \text{ and } m_{n+1} = N(1 - e^{-\frac{m_n}{N}}) \\
P_{success} &\approx 1 - (1 - P_{table})^l \\
M &= m \cdot l \cdot m_0 \\
T_P &= l \cdot m \cdot t \cdot tp_0 \\
T_A &= l \cdot t \cdot \frac{t-1}{2} \cdot ta_0 \\
TLU &= l \cdot t
\end{aligned}
$$

6.3 Attacking Cryptographic Hash Functions

As noted, to find a collision in cryptographic hash functions via exhaustive search has a complexity of $2^{k/2}$. A more efficient attack than an exhaustive search is to utilize the inner structure of a hash function to find a differential trail that finally leads to a collision, as shown in Figure 6.2

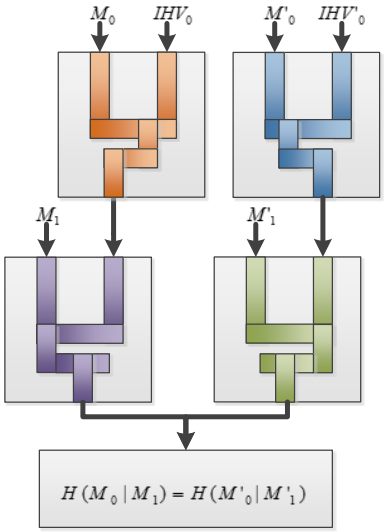


Figure 6.2: Concept of two block collision.

This attack tries to find a collision between two two-block-messages $m = (M_0|M_1)$ and $m' = (M'_0|M'_1)$. For this, two steps are necessary. At first a near collision is generated. That means hashing the first block of messages m and m' leads to an output with a low hamming distance. In the next step the intermediate hash values

(IHV) and the second message block of m and m' are used to generate a full collision. The attacker has to hash the second block of messages m and m' that cancels out previous output differences. In this way the initially assumed $IHV_0 = IHV_0'$ is fulfilled. Figure 6.2 shows the strategy of a two block collision graphically.

6.3.1 MD5 Hash Function Design

In this chapter the design of the MD5 hash function is shown in detail, in order to understand the attacks in more detail.

Merkle-Damgard Design

A cryptographic hash function is made from a one way compression function. One method to design a hash function from a compression function is the Merkle-Damgard (MD) construction, which is used by popular hash functions like SHA-1 SHA-2 and MD5. Basically Merkle-Damgard consists of four main components.

- Message Block Padding (not further considered here)
- Message Block Expansion
- Message Block Compression
- Message Block Mixing

Expansion Function

The message expansion splits the message m into 16 consecutive 32-bit words $b_0, b_1, \cdots, b_{15}$ and expands to 64 32-bit words W_t as follows:

$$W_t = \begin{cases} b_t & \text{for } 0 \leq t < 16, \\ b_{(1+5t) \bmod 16} & \text{for } 16 \leq t < 32, \\ b_{(5+3t) \bmod 16} & \text{for } 32 \leq t < 48, \\ b_{(7t) \bmod 16} & \text{for } 48 \leq t < 64, \end{cases}$$

Compression Function

The compression function involves several rounds with multiple steps. All operations are optimized for 32/64-bit processors with an instruction set that support arithmetic and logic functions. Also very few registers are needed in the hashing process. The compression function is given in Figure 6.3 and defined in the following manner:

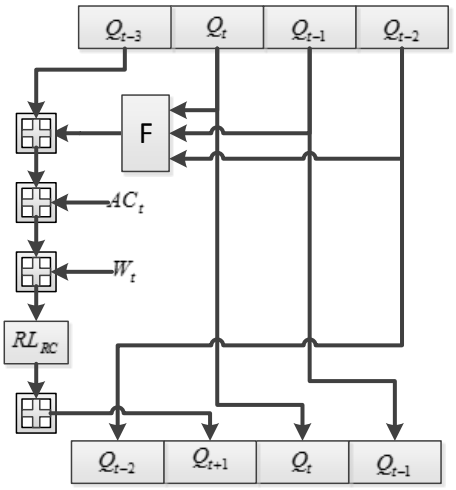
The compression function is working on 4 state register $Q_t, Q_{t-1}, Q_{t-2}, Q_{t-3}$. These are initialized with the IHV.

$$\begin{aligned} F_t &= f_t(Q_t, Q_{t-1}, Q_{t-2}), \\ T_t &= F_t + Q_{t-3} + AC_t + W_t, \\ R_t &= RL(T_t, RC_t), \\ Q_{t+1} &= Q_t + R_t. \end{aligned}$$

With the *Round Function* f_t defined as:

$$f_t(X,Y,Z) = \begin{cases} F(X,Y,Z) = (X \wedge Y) \oplus (\bar{X} \wedge Z) & \text{for } 0 \leq t < 16, \\ G(X,Y,Z) = (Z \wedge X) \oplus (\bar{Z} \wedge Y) & \text{for } 16 \leq t < 32, \\ H(X,Y,Z) = X \oplus Y \oplus Z & \text{for } 32 \leq t < 48, \\ I(X,Y,Z) = Y \oplus (X \vee \bar{Z}) & \text{for } 48 \leq t < 64, \end{cases}$$

Figure 6.3: MD5 compression function.



The *Constant Function* AC_t is computed in every round with the function:

$$AC_t = \lfloor 2^{32} |sin(t+1)| \rfloor, \quad 0 \leq t < 64,$$

Since the function is not round and data dependent they can be easily precomputed.

RL is a parametrized *Left Rotation*. The parameter *RC* is the rotation value and changes every fourth round. The exact parameter values are given as:

$$(RC_t, RC_{t+1}, RC_{t+2}, RC_{t+3}) = \begin{cases} (7,12,17,22) & \text{for } t = 0,4,8,12, \\ (5,9,14,20) & \text{for } t = 16,20,24,28, \\ (4,11,16,23) & \text{for } t = 32,36,40,44, \\ (6,10,15,21) & \text{for } t = 48,52,56,60, \end{cases}$$

Mixing Function

The mixing function mixes the intermediate hash value *IHV* with the next message block. Given an initialization vector $IV = IHV_0$

$$IHV_0 = (a_0, b_0, c_0, d_0)$$

Then the mixing function returns

$$IHV_{i+1} = (a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}) = (a_i + Q_{61}, b_i + Q_{64}, c_i + Q_{63}, d_i + Q_{62})$$

with

$$(Q_{61}, Q_{64}, Q_{63}, Q_{62}) = MD5Compression(M_i)$$

Security Considerations

In the history of MD5 many attacks and algorithms were presented that target the collision resistance property. The most successful examples are listed in the following.

1993 *de Boer and Bossealers* present an algorithm for pseudocollisions (different IV $\rightarrow$ same hash digest)

1996 *Dobbertin* presents first real collision on MD5 variant (specially constructed IV)

2004 *Wang et al.* provide algorithm for systematic collision generation (common-prefix collisions).

2008 *Stevens et al.* present rogue MD5 certificate

6.3.2 Attacks on MD5 Hash Function

A first attack is the construction of a differential trail, which is not very intuitive. Differential trails are mostly hand made (cf. M. Stevens' MA-Thesis). The reverse-engineering of the seminal work which was presented by Wang.

MD5 works on 32-bit unsigned integer values. Each value is defined as a word where each bit will be numbered from the least significant bit 0 to the most significant bit 31. In the following, the notation for this chapter is introduced:

- The word addition $+$ and subtraction $-$ are implemented as an integer addition respectively subtraction modulo 2^{32}.
- $X[i]$ represents the i-th bit of word X
- A binary signed digit representation (BSDR) of a word X is defined as a sequence of digits $X = (k_i)_{i=0}^{31}$ while $k_i \in \{-1, 0, +1\}$ for $0 \leq i \leq 31$. The digits are determined by $X \equiv \sum_{i=0}^{31} k_i 2^i \mod 2^{32}$, e.g. $fc00f000_{16} \equiv= (-1 \cdot 2^{12}) + (+1 \cdot 2^{16}) + (-1 \cdot 2^{26})$.
- The XOR difference is defined as $\Delta X = (k_i)_{i=0}^{31}, k_i = X'[i] - X[i]$.
- The additive difference is defined as $\delta X = X' - X \mod 2^{32}$.
- A difference of the form $+2^j$ is denoted as j^+ and -2^j as j^-.

The goal of this attack is to find a differential trail in the MD5 algorithm, based on XOR and additive differences. In their work, Wang et al. have presented a differential through the MD5 algorithm. This trail is given in Table 6.1. To use this trail to generate a collision, random messages that satisfy the given conditions have to be chosen. The conditions on input messages ensure that the chosen differential trail is taken. The next step is to walk the trail and check if the chosen messages keep walking the trail by holding all conditions. All conditions are given by additive differences where X can be any value.

To ensure that the R_t provides the correct add-difference we consider special conditions on T_t. The conditions fall in one of the following three categories:

- δX *must not* propagate XOR differences past bit position rotated to $R_t[31]$ and thus to lower order bits. This would result in a wrong add-difference in R_t.

- δX *must* propagate differences past bit position $R_t[31]$ to avoid that effects are lost due to 2^{32} reduction.
- δX *must* propagate XOR difference past a certain position

The input differentials for two-block messages $m = (M_0|M_1)$ and $m' = (M_0'|M_1')$ are defined as follows:

$$\Delta H_0 \xrightarrow{(M_0,M_0')} \Delta H_1 \xrightarrow{(M_1,M_1')} \Delta H = 0$$

Define m and m' by

$$\begin{aligned}
\Delta M_0 &= M_0' - M_0 = (0,0,0,0,2^{31},0,0,0,0,0,0,2^{15},0,0,2^{31},0)\\
\Delta M_1 &= M_1' - M_1 = (0,0,0,0,2^{31},0,0,0,0,0,0,-2^{15},0,0,2^{31},0)\\
\Delta H_1 &= (2^{31},2^{31}+2^{25},2^{31}+2^{25},2^{31}+2^{25}).
\end{aligned}$$

The given differences are hand-picked and other solutions are possible as well.

After message expansion, *block* ΔM_0 is expanded to corresponding W_t as follows:

$$\begin{aligned}
W_4' - W_4 = W_{23}' - W_{23} = W_{37}' - W_{37} = W_{60}' - W_{60} &= -2^{31},\\
W_{11}' - W_{11} = W_{18}' - W_{18} = W_{34}' - W_{34} = W_{61}' - W_{61} &= +2^{15},\\
W_{14}' - W_{14} = W_{25}' - W_{25} = W_{35}' - W_{35} = W_{50}' - W_{50} &= -2^{31},
\end{aligned}$$

After message expansion, *block* ΔM_1 is expanded to corresponding W_t as follows:

$$\begin{aligned}
W_4' - W_4 = W_{23}' - W_{23} = W_{37}' - W_{37} = W_{60}' - W_{60} &= -2^{31},\\
W_{11}' - W_{11} = W_{18}' - W_{18} = W_{34}' - W_{34} = W_{61}' - W_{61} &= -2^{15},\\
W_{14}' - W_{14} = W_{25}' - W_{25} = W_{35}' - W_{35} = W_{50}' - W_{50} &= -2^{31},
\end{aligned}$$

The input differentials shall propagate as shown in Table 6.1. The last rounds and steps do not change much in the output. This indicates a weak diffusion behavior in MD5. Finally, we achieve a difference after hashing M_0 and M_0':

$$\Delta H_1 = (2^{31},2^{31}+2^{25},2^{31}+2^{25},2^{31}+2^{25})$$

The collision search can be further improved. There are *290 conditions on first blocks* M_0 and *310 conditions on the second blocks* M_1. There exists different strategies to find messages that satisfies these conditions. Some are mentioned in the following:

- *Sufficient Conditions:* Sufficient conditions (e.g. $Q_t[i] = 0$) guarantee that necessary carries and correct Boolean function differences happen.

- *Early Aborts:* Discard the message candidate when conditions are not satisfied during round 1 (step $0-19$).
- *Multi-Message Modification:* "short-cuts" are defined to modify a message candidate in round > 1 without altering conditions of previous rounds.
- *Tunnels:* precomputed patterns that satisfy a large number of conditions.

In the following a pseudo-algorithm is given to run the search for the first message blocks .

Algorithm 4: Search of Block # 1

1: Choose $Q_1, Q_3, \ldots, Q_{16}$ fulfilling conditions;
2: Calculate $b_0, b_6, \ldots, b_{15}$;
3: **repeat**
4: Choose Q_{17} fulfilling conditions;
5: Calculate b_1 at $t = 16$;
6: Calculate Q_2 and b_2, b_3, b_4, b_5;
7: Calculate $Q_{18}, \ldots, Q_{21}$;
8: **until** $Q_{17}, \ldots, Q_{21}$ are fulfilling conditions
9: **for all** possible Q_9, Q_{10} that satisfying conditions such that b_{11} does not change: (Use tunnels $\mathscr{T}(Q_9, b_{10}), \mathscr{T}(Q_9, b_9)$ and $\mathscr{T}(Q_{10}, b_{10})$) **do**
10: Calculate $b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}$;
11: Calculate $Q_{22}, \ldots, Q_{64}$;
12: Verify condition on $Q_{22}, \ldots, Q_{64}, T_{22}, T_{34}$ and the IHV-conditions for the next block. Stop searching if all conditions are satisfied and a near-collision is verified.
13: **end for**
14: Start again at step 1.

A similar algorithm and strategy with different differential trail for the second block M_1 and M_1' can be used. The overall complexity, including the introduced improvements, is given in Table 6.2. This numbers were measured by M. Stevens et al. (Stevens et al. [2009]).

The intermediate hash value (IHV_0) is used to compress the first message block of the attack. This does not necessarily have to be the MD5 initial value (IV), it could also be the IHV of previous processed blocks. As given in Section 6.3.1 the IHV_{i+1} depends on IHV_i and the result of the compression function of the last processed block. The search algorithm of block 2 has some sufficient conditions on the IHV_1 , which concludes that the complexity and thus the runtime of the attack depends on the initial hash value IHV_0. This can be seen in Table 6.2 as well.

6.3.3 Attack Scenarios

When attacking cryptographic hash functions there are many possible attack scenarios. The most common are shown in the following:

- *Colliding Documents:* Create a pair of files that print different contents in displayable part with hidden tail so that the same MD5 digest is generated (Daum and Licks [2005]) (Gebhardt et al. [2006]).
- *Chosen-Prefix Attack:* A pair of messages with arbitrary chosen prefixes P and P' with constructed suffixes S and S' so that $H(P|S) = H(P'|S')$ (Stevens et al. [2012]).

- *Manipulating MD5 Integrity Checks:* There are several examples in the literature, please see (Stevens et al. [2012]) (Mikle [2004]).
- *Nostradamus Attack:* Commit to a hash value for a given message and construct later another message with the same hash. Can be done by initially constructing colliding message before sending the commitment (Kelsey and Kohno [2006]).

6.3.4 Attack on X.509 Certificates

In July 2008, the RapidSSL CA still issued a majority of MD5-based certificates. Chosen-Prefix attacks on these kinds of certificates were possible. The first task was to ensure the validity of the certificate. That meant to have a valid serial number in (fixed) prefixes that needed to be predicted. The second task was to create the MD5 collision on 15 input blocks with different IHV and IHV'. This took around 3 days on 200-unit PS3 clusters (et al [2008]).

6.4 Tools Supporting Symmetric Attacks

In the following some tools are given, that have implemented the attacks of this chapter.

6.4.1 Tools Utilizing Time-Memory Trade-Off Attacks

TMTO attacks, especially Oechslins Rainbowtables, are widely used. The most common use cases are the cracking of (password) hashes and the popular attack of GSM.

To attack GSM, a hardware sniffer (for instance USRP) is necessary to sniff GSM data. The once sniffed data can be processed with Airprobe and Kraken to decrypt a call.

Airprobe is a tool which captures non-hopping GSM downlink channels with USRP.
(`https://svn.berlin.ccc.de/projects/airprobe`)

Kraken is an open source tool to crack A5/1 keys used to secure GSM 2G calls and SMS. This is done via Rainbow Tables available via torrent.
(`https://opensource.srlabs.de/projects/a51-decrypt`)

A popular target for rainbowtables are password hashes from Windows distributions. Ophcrack and RainbowCrack are tools that are well suited for this job.

Ophcrack uses free available rainbow tables to crack LM and NTLM hashes of Windows distributions. A Live CD to attack a present system on the fly is available as well.
(`http://ophcrack.sourceforge.net`)

RainbowCrack is a full TMTO suite including table generation, sort and lookup. The tool supports GPU acceleration and provides precalculated rainbowtables for LM, NTLM, MD5 and SHA1 hashes.
(`http://www.project-rainbowcrack.com/index.htm`)

6.4.2 Tools MD5 Collision search

In the following some tools are given that implemented the MD5 collision attack.

HashClash is a BOINC initiative to generate MD5 and SHA1 collisions.
(`http://www.win.tue.nl/hashclash/`)

Pack3 generates two archives each with 3 files with colliding MD5 hashes.
(`http://cryptography.hyperlink.cz/2006/selfextract.zip`)

MD5 collision tool that uses tunnels to find collisions in a minute.
(`http://cryptography.hyperlink.cz/2006/web_version_1.zip`)

Table 6.1: The Wang et al. MD5 collision trail for the first message block.

t	δQ_t	δf_t	δQ_{t-3}	δW_t	δT_t	$S(t)$	δR_t
0-3	-	-	-	-	-	-	-
4	-	-	-	$\bar{31}$	$\bar{31}$	7	$\bar{6}$
5	$\bar{6}$	$+19,+11$	-	-	$+19,+11$	12	$+31,+23$
6	$\pm 31,+23,\bar{6}$	$\bar{14},\bar{10}$	-	-	$\bar{15},+14,\bar{10}$	17	$+31,\bar{27},\bar{0}$
7	$\bar{27},+23,\bar{6},\bar{0}$	$\bar{27},\bar{25},+16,+10,5,\bar{2}$	-	-	$\bar{27},\bar{25},+16,+10,5,\bar{2}$	22	$+27,\bar{24},\bar{17},\bar{15},+6,+1$
8	$\bar{23},\bar{17},\bar{15},+0$	$\pm 31,\bar{24},+16,+10,+8,+6$	$\bar{6}$	-	$\bar{31},\bar{24},+16,+10,+8$	7	$\bar{31},+23,+17,\bar{15},6$
9	$\pm 31,\bar{6},+0$	$\pm 31,+26,\bar{23},\bar{20},+6,+0$	$\pm 31,+23,\bar{6}$	-	$+26,\bar{20},+0$	12	$+12,+6,\bar{0}$
10	$+31,+12$	$\bar{23},+13,+6,+0$	$\bar{27},+23,\bar{6},\bar{0}$	-	$\bar{27},+13$	17	$+30,\bar{12}$
11	$+31,+30$	$\bar{8},\bar{0}$	$\bar{23},\bar{17},\bar{15},+0$	$+15$	$\bar{23},\bar{17},\bar{8}$	22	$\bar{30},\bar{13},\bar{7}$
12	$+31,\bar{13},\bar{7}$	$+31,+17,+7$	$\bar{31},6,+0$	-	$+17,+6,+0$	7	$+24,+13,+7$
13	$+31,+24$	$+31,\bar{13}$	$+31,+12$	-	$\bar{12}$	12	$\bar{24}$
14	$+31$	$+31,+18$	$+31,+30$	$\bar{31}$	$\bar{30},+18$	17	$\bar{15},+3$
15	$+31,\bar{15},+3$	$+31,+25$	$+31,\bar{13},\bar{7}$	-	$+25,\bar{13},\bar{7}$	22	$\bar{29},+15,\bar{3}$
16	$+31,\bar{29}$	$+31$	$+31,+24$	-	$+24$	5	$+29$
17	$+31$	$+31$	$+31$	-	-	9	-
18	$+31$	$+31$	$+31,\bar{15},+3$	$+15$	$+3$	14	$+17$
19	$+31,+17$	$+31$	$+31,\bar{29}$	-	$\bar{29}$	20	$\bar{17}$
20-21	$+31$	$+31$	$+31$	-	-	-	-
22	$+31$	$+31$	$+31,+17$	-	$+17$	14	$+31$
23	-	-	$+31$	$\bar{31}$	-	20	-
24	-	$+31$	$+31$	-	-	5	-
25	-	-	$+31$	$\bar{31}$	-	9	-
26-33	-	-	-	-	-	-	-
34	-	-	-	$+15$	$+15$	16	$+31$
35	$+31$	$\bar{31}$	-	$\bar{31}$	-	23	-
36	$+31$	-	-	-	-	4	-
37	$+31$	$+31$	-	$\bar{31}$	-	11	-
38-49	± 31	± 31	± 31	-	-	-	-
50	$\bar{31}$		$\bar{31}$	$\bar{31}$	-	11	-
51-59	$\bar{31}$	$+31$	± 31	-	-	-	-
60	$+31$		$\bar{31}$	$\bar{31}$	-	6	-
61	$\bar{31}$	$\bar{31}$	$\bar{31}$	$+15$	$+15$	10	$+25$
62-63	$\pm 31,+25$	± 31	± 31	-	-	-	-

Table 6.2: Complexity of MD5 Attack on a Pentium 4 @ 3 GHz.

	Avg. Time (s)	Max. Time (s)	Avg. Complex.	# Tests
MD5 IV	64	428	$2^{32.25}$	1048
recommended IV	67	660	$2^{32.33}$	1138
random IV's	291	15787	$2^{34.08}$	879

Note: Average complexity means the average number of MD5 block compressions.

6.5 Workshop: Time-Memory Trade-Off Attacks

Background: You are a security consultant and requested to perform a security audit on hardware security tokens manufactured by a company named *Security-Masters* (SM). The company praises their tokens to provide bullet-proof two-factor authentication based on one-time-passwords for online-banking applications (ANSI X9.9). The hardware security token is shown in Figure 6.4.

Figure 6.4: Hardware Security Token

The token is equipped with a display that present 9 alphanumerical numbers to the user. Due to this limitation all tokens are seeded with a PIN of 9 hexadecimal digits from which the three subkeys K_1, K_2, K_3 for a 3DES block cipher are derived[1]. Key derivation works as follows: the PIN is copied to the least significant hex digits of K_1, all other digits are set to zero. Subkey K_2 and K_3 are generated internally by shifting K_1 by 5 and 23 bits to the left: $K_2 = S_5(K_1)$ and $K_3 = S_{23}(K_1)$. A monotonic non-volatile counter is attached to the plaintext input of the 3DES cipher which is initialized to zero. Every invocation of the token increases the counter by one. You are finally told that during test and quality check each token is invoked exactly 100 times to make sure that the device operates correctly. The design of the security system inside the token is depicted in Figure 6.5.

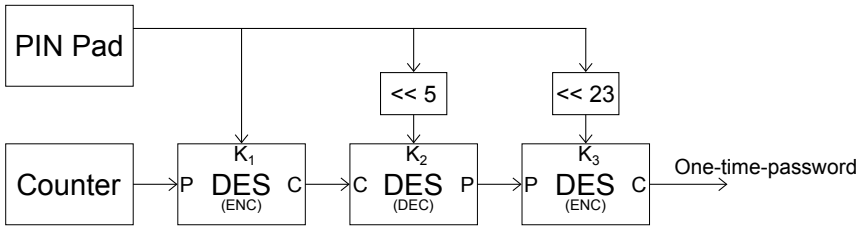


Figure 6.5: Structure of the Token

The chief security officer of SM provided you with 100 new hardware tokens that just came from of the quality check. Your task is to provide an exhaustive security audit within one week. In case of a security weakness, you are asked to backup your findings with a proof-of-concept attack (at best by extracting the keys from the devices and cloning tokens).

You are now asked to complete the following tasks.

Exercise 6.1

Give a statement why Time-Memory Trade-Offs can be suitable way to attack the hardware token system.

Ü

[1] Note that for the key initialization the buttons `0-5` can be switched to alpha mode `A-F`.

Ü

Exercise 6.2

Provide suitable parameters for a rainbow table-based TMTO to successfully clone at least 80 devices.

Ü

Exercise 6.3

Implement an application to generate precomputation tables using the following binary format. Each table must be placed in a separate file. Each table files contain consecutive chain entries without any delimiters. Each chain entry consists of 16 byte. The first 8 byte of a chain entry represent the start point (SP) while the last 8 byte the corresponding end point (EP), both in little endian notation. Development of the tool can be done in the programming language of your choice.

Ü

Exercise 6.4

Run the precomputation application to generate the tables. Then use the provided tool to sort your tables.

Ü

Exercise 6.5

Implement a further application to conduct the attack-phase based on your precomputation data.

Ü

Exercise 6.6

Download the list with one-time-passwords (corresponding to your group) from the Blackboard. Run the attack and recover PINs for at least 50% of the devices to pass this task and complete your security audit.

6.6 Lessons Learned

The following items have been discussed as part of this chapter:

- Time-Memory Trade-Off attacks partially moves the time complexity of an attack into the precomputation phase which drastically reduces the run time of the attack phase. Therefore, TMTO attacks are only reasonable to attack multiple targets due to a single executed precomputation phase.
- MD5 collisions can be generated in seconds. A prefix-attack on MD5-based certificates in 2009 demonstrated that MD5 should not be employed in any cryptographic settings anymore.

Chapter 7 Tools for Asymmetric Cryptanalysis

For advanced security services, such as key agreement over an untrusted network, asymmetric cryptosystems are essential. The recently widely used asymmetric cryptosystems in practice are based on two mathematical problems that are assumed to be hard. Both fundamental mathematical problems, however, can be attacked by solvers. This chapter introduces the corresponding solver algorithms for (a) the integer factorization problem as used for the RSA assumption and (b) the discrete logarithm problem that is included in the construction of the Diffie-Hellman Key Exchange, ElGamal encryption but also Elliptic Curve Cryptosystems.

7.1 Integer Factorization Problem

7.1.1 Background

Factorization is an important mathematical problem with a long history dating back to ancient Greece (Sieve of Eratosthenes). Throughout history, many well-known mathematicians like Fermat, Euler, and Gauss worked on factorization and proposed algorithms to solve it. With the invention of modern public key cryptography (RSA), this field of research became more important resulting in plenty of new attacks invented in the last 30 years.

The problem can be described as follows:
Given a composite number $N = \prod_i p_i^{e_i}$ consisting of unknown primes p_i and exponents $e_i \in \mathbb{N}$. The attacker's task is to find the prime factors p_i and exponents e_i.

Although factorization is a problem with a long history, its actual difficulty is still unclear. The decision problem version ("Has N a factor less than M?") is known to be in NP and co-NP. However, it is further assumed neither to be NP-complete or co-NP-complete. In particular, factorization can be efficiently solved with quantum computers. Shor's algorithms can find the factorization of an integer in polynomial time (Shor [1997]) so that RSA-based cryptosystems will become insecure as soon powerful quantum computers turn into reality. For more details about NP we want to refer the reader to (Blömer [2006]).

Preliminaries

For a detailed understanding of the factorization problem and the techniques used in corresponding solvers, we now introduce a few essential theorems on prime numbers.

Definition 7.1: Prime Numbers

A prime number is a natural number $p \in \mathbb{N}, p > 1$ with no other divisors than 1 and itself. $\mathbb{P}$ is the set of all primes.

D

It is further important to know how many prime number exist in a given interval of integer. For cryptographic purposes this is directly linked to the question how likely it is to find quickly a prime with n bits (e.g., for RSA) by choosing large random integers. The solution to this question is provided by the following theorem:

D

Definition 7.2: Prime Number Theorem

The number of primes in the interval $[1,x]$ is given by

$$\pi(x) = |\{p \in \mathbb{P} | p \leq x\}|$$

and can be approximated by

$$\lim_{x\to\infty} \frac{\pi(x)}{\frac{x}{ln(x)}} = 1 \Leftrightarrow \pi(x) \approx \frac{x}{ln(x)}.$$

The complexity of factorization can be distinguished into several cases. For example, it might be the case that it is known a priori that N has just two factors or that all factors are below a given threshold:

D

Definition 7.3: Smoothness

Let B be a positive integer and $N = \Pi p_i^{e_i}$. We call N B-smooth if it holds that $p_i \leq B \; \forall i$.

For some cases it is useful to have an even tighter criterion for the smoothness of an integer B.

D

Definition 7.4: Power-Smooth

Let B be a positive integer and $N = \Pi p_i^{e_i}$. We call N B-power-smooth if $p_i^{e_i} \leq B \; \forall i$.

At this point we have a look at an example for a better intuition of the concept:

B

Example 7.1: 5-Power-Smooth number

$30 = 2 \cdot 3 \cdot 5 \Rightarrow 30$ is power-smooth for $B = 5$

Finally we need to distinguish between the solver types for the factorization problem. We classify those according to their complexity either related to the size of the integer to be factored or its smallest factor.

D

Definition 7.5: General-Purpose and Special-Purpose Factoring Algorithms

Factoring algorithms can be divided into two categories (Menezes et al. [1996]).

- The runtime of a *special-purpose* factoring algorithm depends on the size of the smallest prime (or other certain properties) of N.
- The runtime of a *general-purpose* factoring algorithm depends only on the size of the integer to be factorized.

Finally, for very complex algorithms it is handy to have a shorthand for its complexity function. We now introduce the *L*-Notation for this purpose.

D

Definition 7.6: *L*-Notation

In computational number theory, the complexity of algorithms is sometimes described in the *L*-Notation. This notation makes the analysis and comparison of different algorithms easier, as it simplifies the expression. It is defined as

$$L_n[\alpha, c] \;=\; e^{(c+o(1))(ln(n))^{\alpha}(ln(ln(n)))^{1-\alpha}}$$

with $c \in \mathbb{Q}^+$, $0 \leq a \leq 1$ (Lenstra and Jr. [1990]).

This notation is very useful, for example, to characterize the complexity of the best known attack on the integer factorization problem on conventional computing platforms.

B

Example 7.2: General Number Field Sieve

The runtime of the General Number Field Sieve is defined as

$$L_n[\tfrac{1}{3}, c] \;=\; e^{(c+o(1))(ln(n))^{\frac{1}{3}}(ln(ln(n)))^{\frac{2}{3}}}$$

with $c = \sqrt[3]{(\frac{64}{9})}$ (Pomerance [1996]).

We now have a look at the different strategies to solve the integer factorization problem.

Naive Approach using Trial Division

An easy and naive approach to get the factors of a number is trial division. The algorithm divides N consecutively by all natural numbers $n \in \mathbb{N}$ with $n \leq \sqrt{N}$. The complexity of this algorithm is $\mathcal{O}(\sqrt{N})$.

Simple Congruential Method

The simple congruential method assumes a composite N with a small factor d. Given $x_i, x_j \in \mathbb{Z}_N$ with $x_i \neq x_j \pmod N$, it is quite probable that

$$\begin{aligned} & x_i \equiv x_j \pmod d \\ \Leftrightarrow\; & (x_i - x_j) \equiv 0 \pmod d. \end{aligned}$$

Based on this, it is possible to conclude that $d \mid (x_i - x_j)$ but $N \nmid (x_i - x_j)$. So that $gcd(x_i - x_j, N) = d$ gives a non-trivial factor of N. The difficult part in this approach is to find suitable x_i and x_j without knowing d.

B

Example 7.3: Simple Congruential Method

Given: $N = 21, [d = 3]$ (unknown)
Assume

$$x_i = 17 \; [\equiv 2 \pmod 3],$$
$$x_j = 8 \; [\equiv 2 \pmod 3].$$

Then $(x_i - x_j) = 9$ is divisible by 3 but not by 21.

$$\Rightarrow gcd(x_i - x_j, N) = gcd(9, 21) = 3 = d.$$

Congruence of Squares

For congruence of squares it is assumed to have a composite N and two numbers $x_i, x_j < N, x_i \not\equiv \pm x_j \pmod N$ such that

$$x_i^2 \equiv x_j^2 \pmod N$$
$$\Leftrightarrow (x_i^2 - x_j^2) \equiv 0 \pmod N$$
$$\Leftrightarrow (x_i - x_j)(x_i + x_j) = k \cdot N$$

Hence $gcd(x_i - x_j, N)$ and $gcd(x_i + x_j, N)$ reveal non-trivial factors of N.
The difficult task is still to find suitable numbers for x_i and x_j.

B

Example 7.4: Congruence of Squares

Given: $N = 21$
Assume

$$x_i = 17 \Rightarrow x_i^2 \equiv 16 \pmod{21},$$
$$x_j = 11 \Rightarrow x_j^2 \equiv 16 \pmod{21}.$$

Then N can be factorized by:

$$\Rightarrow gcd(x_i - x_j, N) = gcd(17 - 11, 21) = 3$$
$$\Rightarrow gcd(x_i + x_j, N) = gcd(17 + 11, 21) = 7.$$

In the next step we describe the different methods for factorization in detail.

7.1.2 Sieve of Eratosthenes

This method is the oldest and simplest algorithm for factorization (300 BC) and becomes very slow for large number. Nevertheless, it is still the best algorithm for "small" number (e.g., $p < 10^7$). The basic idea of the algorithm is to mark the multiples of primes numbers starting from $p = 2$. Figure 7.1 shows an example of the sieve for numbers up to 50.

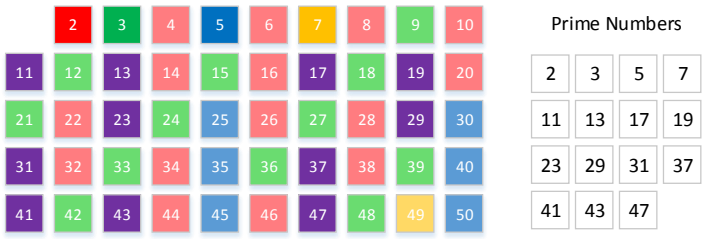


Figure 7.1: Application of the sieve of Eratosthenes for the number up to 50.

7.1.3 Pollard's p-1 Method

Pollard's p-1 method assumes a composite N with a prime $p \mid N$. The first step is to choose $a \in \mathbb{Z}_n$ with $gcd(a,N) = 1$. The method is based on Fermat's little theorem:

$$a^{p-1} \equiv 1 \pmod{p}.$$

The k-th extension of the equation is defined as

$$\begin{aligned}(a^{p-1})^k &\equiv 1^k \pmod{p} \\ a^{k(p-1)} &\equiv 1 \pmod{p}.\end{aligned}$$

Then $k(p-1)$ is set to e and it can be concluded that

$$\begin{aligned}a^{k(p-1)} = a^e &\equiv 1 \pmod{p} \\ \Leftrightarrow a^e - 1 &\equiv 0 \pmod{p} \\ \Rightarrow p|(a^e - 1).\end{aligned}$$

Hence if $a^e \neq 1$, a non-trivial factor can be computed by $gcd(a^e - 1, N)$.

The difficult task of this algorithm is finding e without knowing p. In order to solve this problem it is assumed that $p-1$ is B-power-smooth (consists only of small primes). Then the exponent can be computed by $e = \prod_{p_i \in \mathbb{P}, p_i \leq B} p_i^{\lfloor log_{p_i}(B) \rfloor}$.

If B is chosen too small, then no factor of N will be B-power-smooth and $gcd(a^e -$

$1,N)$ returns 1. If B is chosen too big, all factors of N will be B-power-smooth and $a^e - 1 \pmod N$ will be zero.

B

Example 7.5: Factoring using Pollard's p-1 Method

Given: $N = 5917, B = 5, a = 2$
Compute e:

$$e = lcm(1,2,3,4,5) = 60$$

Compute $a^e - 1$:

$$2^{60} - 1 \equiv 3416 \pmod{5917}$$

Find a factor of N:

$$\Rightarrow gcd(2^{60} - 1, 5917) = gcd(3416, 5917) = 61$$

Finally we yield $5917 = 61 \cdot 97$

7.1.4 Elliptic Curve Method

This method is conceptually identical to Pollard's p-1 method. It shifts the computations from the integers to the "group" on elliptic curves:

$$q = a^e - 1 \equiv 0 \pmod p \Rightarrow gcd(q,N) = d$$
$$e \cdot P = Q \Rightarrow gcd(\mathbb{Z}_Q, N) = d.$$

The benefit is that if the computation fails, i.e. a trivial factor is computed, it can be restarted with a different elliptic curve, i.e. with a different group order $ord(E)$. Additionally, the algorithm is extended by a second phase which helps to cover "large" factors. Algorithm 5 shows all the steps necessary for the elliptic curve method.

Algorithm 5: The Elliptic Curve Method

Input: Composite $n = f_1 \cdot f_2 \cdot \cdots \cdot f_n$
Output: Factor f_i of n

Phase 1:
Choose arbitrary curve $E(\mathbb{Z}_n)$ and random point $P \in E(\mathbb{Z}_n) \neq O$
Choose smoothness bounds $B1, B2 \in \mathbb{N}$.
Compute $e = \prod_{p_i \in \mathbb{P}; p_i < B1} p_i^{\lfloor log_{p_i}(B1) \rfloor}$
Compute $Q = eP = (x_Q; y_Q; z_Q)$.
Compute $d = gcd(z_Q, n)$.
Phase 2:
Set $s := 1$.
for each prime p with $B1 < p \leq B2$ **do**
    Compute $pQ = (x_{pQ}; y_{pQ}; z_{pQ})$.
    Compute $d = d \cdot z_{pQ}$.
end for
Compute $f_i = gcd(d, n)$.
if $1 < f_i < n$ **then**
    A non-trivial factor d is found.
    return f_i
else
    Restart from Step 2 in Phase 1
end if

7.1.5 Pollard's Rho Algorithm

This algorithm can be found in the next section (7.2.3), since it is exactly the same for the discrete logarithm problem.

7.1.6 Sieving Methods

The general concept of this method is as follows. At first, a factor base $F = \{p_1, p_2, \ldots, p_m\}$ containing distinct primes p_i is found. Then a set of integers $U = \{r_1, ..., r_n\}$ is constructed such that

$$\forall i \ : \ f(r_i) \ = \ r_i^2 \ = \ p_1^{e_{i1}} \cdot p_2^{e_{i2}} \cdot \cdots \cdot p_m^{e_{im}} \pmod{m}.$$

In other words, all $f(r_i)$ have to be smooth over the factor base F. Additionally, U must have the second property that

$$\forall i, 1 \ \leq \ j \ \leq \ m \ : \ \textstyle\sum_{i=1}^{n} e_{i,j} \ = \ 2 \cdot e_i \text{ for some } e_i \ \in \ \mathbb{Z}.$$

Then it is possible to rewrite

$$\textstyle\prod_{i=1}^{n} f(r_i) \ = \ \prod_{i=1}^{n} r_i^2 \ = \ (p_i^{e_1}, p_i^{e_2}, \ldots, p_i^{e_m})^2 \pmod{N}$$

Finally the products

$$x_i \ = \ \textstyle\prod_{i=1}^{n} r_i \text{ and } x_j \ = \ \prod_{i=1}^{m} p_i^{e_i}$$

can be used to construct a congruence of squares for factorization. The main problem of this algorithm is the construction of a set U with these particular properties.

One approach is to find more than m integers r_i with $f(r_i) = p_1^{e_{i1}} \cdot \cdots \cdot p_m^{e_{im}}$. Then a subset of these integers is picked which fulfills the second property. This is usually done by representing each r_i as an m-dimensional vector consisting of the exponents $(e_{i1}, \ldots, e_{im})$. Now the task is to find a set of these vectors for which the sum modulo 2 is the zero vector (Bimpikis and Jaiswal [2005]). In the following example this procedure is explained in more detail.

The process described above is only the basic structure for sieving algorithms. There are different sieving algorithms which adjust this structure to achieve better performances. Some of them are:

- **Quadratic Sieve:** $f(r_i) = r_i^2 \bmod N$ is replaced by $f(r_i) = r_i^2 - N$ for efficiency reasons. This accelerates the checking for smoothness of potential r_i.
- **Special Number Field Sieve:** This approach uses special rings instead of integer rings.
- **General Number Field Sieve:** This approach uses rings over rational numbers instead of integer rings. It is the fastest known method for factorization with a complexity of $L[1/3, (64/9)^{1/3}]$ (Pomerance [1996]).

B

Example 7.6: Quadratic Sieve

In this example, $N = 87463$ is factorized using the quadratic sieve. The advantage of the quadratic sieve over the basic algorithm lies in the adjusted f-function.

In the basic algorithm, trial division is used to test the smoothness of every potential $f(r_i)$. By changing the function to $f(x) = x^2 - N$, it is possible to take advantage of the following equation

$$\begin{aligned} f(x+kp) &= (x+kp)^2 - N \\ &= x^2 + 2xkp + (kp)^2 - N \\ &\equiv x^2 - N \pmod{p} \equiv f(x) \end{aligned}$$

This means that the value of $f(x+kp)$ for $k \in \mathbb{N}$ is equivalent to $f(x)$ modulo p. From this, it can be concluded that

$$\begin{aligned} f(x) \equiv 0 \pmod{p} \equiv f(x+kp) \\ \Rightarrow p \mid f(x) \textit{ and } p \mid f(x+kp) \end{aligned}$$

That means if p divides $f(x)$, it consequently divides all $f(x+kp)$ as well. Therefore, if the roots of the equation $x^2 \equiv N \pmod{p}$ are found, then automatically all numbers are found which are divisible by p. Thus eliminating the process of trial-division.

To make sure these roots exist, a factor base with special attributes has to be selected. For each p_i, $x^2 = N \bmod p_i$ needs to have a solution, which means N has to be a quadratic rest modulo p_i. This can be checked using the Legendre symbol.

To find an appropriate F, the Legendre symbol $\left(\frac{n}{p_i}\right)$ for all prime numbers p_i in a certain range is computed (Table 7.1).

All prime numbers for which the Legendre symbol is 1 can be used for the algorithm. Additionally, -1 is also included in F to allow r_i with a negative $f(r_i)$. The factor base in this example is then $F = \{-1, 2, 3, 13, 17, 19, 29\}$.

p_i	**2**	**3**	5	7	11	**13**	**17**	**19**	23	**29**	31	37
$\left(\frac{n}{p_i}\right)$	1	1	-1	-1	-1	1	1	1	-1	1	-1	-1

Table 7.1: Computation of the Legendre symbol for various prime numbers

The next step is to find all roots α, β of N modulo p (Table 7.2).

p	2	3	13	17	19	29
α, β	1	1,2	5,8	7,10	5,14	12,17

Table 7.2: The roots of N modulo each p_i.

Now the sieving process can start. First an array containing all $f(r_i)$ for a set of integers in a chosen interval is created. In this example the interval is $[\lfloor\sqrt{N}\rfloor - 30, \lfloor\sqrt{N}\rfloor + 30] = [265, 325]$. The process is shown for the first four integers. At the start, the array contains $[f(265), f(266), f(267), f(268)]$ (Table 7.3).

$f(265)$	$f(266)$	$f(267)$	$f(268)$
-17238	-16707	-16174	-15639

Table 7.3: Array the start of the sieving process.

In the first step, all negative numbers are divided by -1. The result is shown in Table 7.4.

Then for the next prime $p_2 = 2$ and $\alpha = 1$, all array positions equal to α or $\alpha + kp_2$ are divided by p_2 and its powers. The result is shown in Table 7.5.

This is done for all $p_i \in F$ and their corresponding α and β. At the end, all array position which hold a 1 are smooth over F (Table 7.6).

In this example, this means that $f(265)$ is smooth over F while the others are not.

Table 7.4: Array after the first step.

$f(265)$	$f(266)$	$f(267)$	$f(268)$
17238	16707	16174	15639

Table 7.5: Array after the second step.

$f(265)$	$f(266)$	$f(267)$	$f(268)$
8619	16707	8087	15639

The next step is to create the matrix A consisting of the exponent vectors for all found r_i modulo 2 (Table 7.7).

Using A, the task is to find a solution $\underline{v}$ for the equation:

$$A^T \underline{v} \;=\; \underline{0} \pmod 2$$

The more integers r_i are used in this step, the more probable it is to find a solution. In this example the equation is:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \cdot \underline{v} \;=\; \underline{0} \pmod 2$$

One possible solution is $\underline{v} = (1,1,1,0,1,0)$. Therefore, the subset which fulfills both properties is $U = \{265, 278, 296, 307\}$. With this x_i, x_j can be computed as follows:

$$x_i = 265 \cdot 278 \cdot 296 \cdot 307 \equiv 34757 \pmod{87463}$$
$$x_j = \sqrt{(265^2 - N) \cdot (278^2 - N) \cdot (296^2 - N) \cdot (307^2 - N)} \equiv 28052 \bmod 87463$$

Now N can be factorized by computing:

$$gcd(x_i - x_j, N) = gcd(6705, 87463) = 149$$
$$gcd(x_i + x_j, N) = gcd(62809, 87463) = 587$$

7.2 Discrete Logarithm Problem

7.2.1 Background

Just as factorization, the task to solve discrete logarithms efficiently is a crucial and established mathematical problem in cryptography.

The problem can be described as follows:
Given a cyclic group of order m and a primitive element $g \in \mathbb{G}$. In multiplicative notation, we define the k-fold application of the group operation as mapping:

$$exp_g : \mathbb{G} \to \mathbb{G} : k \mapsto g^k = \underbrace{g \circ g \circ g \circ g \circ \cdots \circ g}_{\text{k times}}$$

$f(265)$	$f(266)$	$f(267)$	$f(268)$
1	5569	8087	401

Table 7.6: Array after all $p_i \in F$ are processed.

r_i	**-1**	**2**	**3**	**13**	**17**	**19**	**29**
265	1	1	1	0	1	0	0
278	1	0	1	1	0	0	1
296	0	0	0	0	1	0	0
299	0	1	1	0	1	1	0
307	0	1	0	1	0	0	1
316	0	0	0	0	1	0	0

Table 7.7: The exponent vectors for all r_i.

The attacker's task is to find the inverse mapping $log_g = exp_g^{-1}$ to determine k given x and g so that $x = g^k$.

Remark that the problem is not difficult in all groups (cf. the specified levels of security in Section 3.1), e.g.:

- $(\mathbb{Z}_p, t) : g + g + \cdots + g = k \cdot g = x \rightarrow k = x \cdot g^{-1}$ (EEA).
- $(\mathbb{Z}_p^*, \cdot) : g^k \pmod p \rightarrow$ hard for $p > 1024$ bits.
- $(\mathbb{E}, P_{ADD}) : k \cdot G \rightarrow$ hard for $ord(E) > 160$ bits.

Several attacks used for factorization can also be applied to solve discrete logarithms. Similar to factorization, there exists an algorithm to solve discrete logarithms in polytime on quantum computers (Shor [1997]).

Preliminaries

D

Definition 7.7: Subgroup

Given a cyclic group G of order m and a primitive element g. For each divisor $k|m$ there exists exactly one subgroup $G(k) < G$:

1. $G(k) = \{x \in G : x^k = e\}$
2. $G(k) = Im(G \xrightarrow{\phi} G)$ with $\phi(x) = x^{\frac{m}{k}}$

Theorem: Given a cyclic group G of order m and $m = p_1 \cdot p_2 \cdot \cdots \cdot p_r$ with distinct, co-prime $p_i \geq 2$. Let $G(p_i) < G$ be the subgroup of G and $\lambda_1, \lambda_2, \ldots, \lambda_r$ integers so that $\lambda_1 \cdot \frac{m}{p_1} + \cdots + \lambda_r \cdot \frac{m}{p_r} = 1$. Then it holds that the maps

$$\phi(x) \quad = \quad (x^{\frac{m}{p_1}}, \ldots, x^{\frac{m}{p_r}})$$

and

$$\psi(x) \quad = \quad x_1^{\lambda_1} \cdot x_2^{\lambda_2} \cdot \cdots \cdot x_r^{\lambda_r}$$

lift elements from G into the set of subgroups

$$G \xrightarrow{\phi} G(p_1) \times G(p_2) \times \cdots \times G(p_r) \xrightarrow{\psi} G.$$

Naive Search

This search is analogous to the trial division for factorization. To find the discrete logarithm of $g^k = x$, it computes $g^{k'} = x'$ for all possible k' until $x' = x$. The complexity of this algorithm is $\mathcal{O}(m)$ and depends on the order m of the group.

Collision Search

The goal of this approach is to construct a collision so that

$$x^d \;=\; g^{kd} \;=\; g^{tu}$$

with known d, t, u. Then k can be computed with $k = \frac{tu}{d}$. The complexity is $\mathcal{O}(\sqrt{m})$ due to the birthday paradox.

Subgroup Search

This approach assumes that the order $m = p_1 \cdot \cdots \cdot p_r$ does not contain a large prime factor p_i. The first step of the algorithm is to factor m. After that, the discrete logarithm has to be solved in each subgroup $G(p_i)$. This is utilized in the Pohlig-Hellman Attack (Pohlig and Hellman [1978]). The complexity of this approach is $\mathcal{O}(\sqrt{max(p_i)})$

In the next step we describe the different methods for solving discrete logarithms in detail. Recall that the task is to find k with $g^k = x$ with a known g, x.

7.2.2 Shank's Baby-Step-Giant-Step Algorithm

Instead of trying all possible k' (see 7.2.1), the Baby-Step-Giant-Step Algorithm proposed by D. Shank in 1971 performs a meet-in-the middle attack. At the beginning, the search space is partitioned into parts of size g^r with $r = \sqrt{m}$ (giant steps). Then the algorithm tries to hit a partition using baby steps. This process is shown in Figure 7.2.

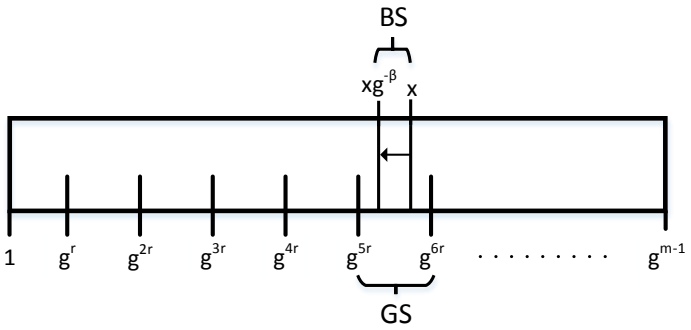


Figure 7.2: Graphical representation of the Baby-Step-Giant-Step algorithm (neglecting the mod operation).

The main idea is to rewrite $k = \alpha r + \beta$ with $0 \leq \alpha, \beta < r$. This concludes

$$g^{\alpha r} \;=\; xg^{-\beta}.$$

The first step is to compute the giant steps $g^{\alpha r} = (g^r)^\alpha$ for $\alpha \in \{0,1,2,\ldots,r-1\}$. Then the results $(\alpha, g^{\alpha r})$ are stored in a table and sorted.

The second step is to find a collision using baby steps. Therefore, $xg^{-\beta}$ for $\beta \in \{0,1,2,\ldots r-1\}$ is computed and compared against the table entries from the first step.

When a collision is found, the discrete logarithm is $k = \alpha r + \beta$.

The complexity of this approach is $O(\sqrt{m})$ in time and space.

Example 7.7: Discrete Logarithm in $\mathbb{Z}_p$

Given: $p = 101, r = 11, g = 2, x = 3$

First compute the giant steps $g^{\alpha r}$.

α	1	2	3	4	5	6	7	8	9	10	11
$g^{\alpha r}$	28	77	35	71	69	13	**61**	92	51	14	89

Then compute the baby steps $xg^{-\beta}$. To improve the performance of the algorithm, the baby steps are changed to xg^{β}. This way the inversion for every step is avoided. Additionally, the formula for k needs to be adjusted to $k = \alpha r - \beta$.

β	1	2	3	4	5	6	7	8	9	10	11
xg^{β}	6	12	24	48	96	91	81	**61**	21	42	8

$\Rightarrow k = \alpha r - \beta = 7 \times 11 - 8 = 69$

B

7.2.3 Pollard's Rho Algorithm

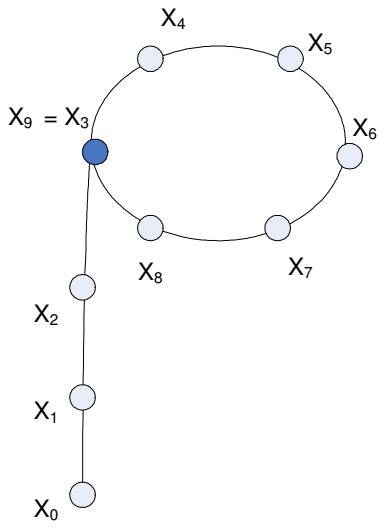


Figure 7.3: Collision path of pseudo-random walk

The idea of this algorithm is to perform a random walk through some finite set S until a collision is found. Then it uses this collision to compute the discrete logarithm. The search path generated by this walk can be seen in Figure 7.3.

To perform the random walk, an iteration function is defined. The basic idea is to partition the search space into n shares s_i (originally $n = 3$). Then the recursive step ($n = 3$) for each iteration is defined as:

$$w_0 = 1 \text{ and } w_{i+1} = \begin{cases} x \cdot w_i & \text{if } w_i \in S_1, \\ w_i^2 & \text{if } w_i \in S_2, \\ g \cdot w_i & \text{if } w_i \in S_3, \end{cases}$$

This means that in each step, the current value is either squared or multiplied by x or g.

The desired collision should have the form $x^{e_1} g^{f_1} = x^{e_2} g^{f_2}$. So it is necessary to update the exponents e and f of w in each step. Since the operation of the recursive step is either squaring or multiplication the coefficients e_i and f_j can be computed with:

$$e_0 = 0 \text{ and } e_{i+1} = \begin{cases} e_i + 1 & \text{if } w_i \in S_1, \\ 2 \cdot e_i & \text{if } w_i \in S_2, \\ e_i & \text{if } w_i \in S_3, \end{cases}$$

$$f_0 = 0 \text{ and } f_{i+1} = \begin{cases} f_i & \text{if } w_i \in S_1, \\ 2 \cdot f_i & \text{if } w_i \in S_2, \\ f_i + 1 & \text{if } w_i \in S_3, \end{cases}$$

With this it is possible to find a collision in time complexity $\mathcal{O}(\sqrt{m})$ and, if all intermediate values are stored, in space complexity $\mathcal{O}(\sqrt{m})$. Therefore, it would not be an improvement to the Baby-Step-Giant-Step algorithm.

To avoid storing all previous values, Floyd's cycle finding algorithm (Knuth [1981]) is applied. It is based on two parallel walks: (w_i, e_i, f_i) and (w_{2i}, e_{2i}, f_{2i}). So in each iteration the first walk makes one step, while the second walk makes two steps. In this approach only $(w_i, e_i, f_i, w_{2i}, e_{2i}, f_{2i})$ has to be stored, making the space requirements negligible (van Oorschot and Wiener [1999]).

When the collision is found, k can be computed as

$$\begin{aligned} w_i = w_{2i} = x^{e_1} \cdot g^{f_1} &= x^{e_2} \cdot g^{f_2} \\ \Leftrightarrow x^{e_1 - e_2} &= g^{f_2 - f_1} \\ \Leftrightarrow g^{k(e_1 - e_2)} &= g^{f_2 - f_1} \\ \Leftrightarrow k &= \frac{f_2 - f_1}{e_1 - e_2}. \end{aligned}$$

B

Example 7.8: Pollard's Rho

Given: $P = 1019, g = 2, x = 5$
For each step i compute (w_i, e_i, f_i) and (w_{2i}, e_{2i}, f_{2i}) until a collision is found.

i	w_i	f_i	e_i	w_{2i}	f_{2i}	e_{2i}
1	2	1	0	10	1	1
2	10	1	1	100	2	2
3	20	2	1	1000	3	3
4	100	2	2	425	8	6
5	200	3	2	436	16	14
6	1000	3	3	284	17	15
7	981	4	3	986	17	17
8	425	8	6	194	17	19
...	...	...	...	...	...	...
48	224	680	376	86	299	412
49	101	680	377	860	300	413
50	505	680	378	101	300	415
51	**1010**	681	378	**1010**	301	416

Then compute k:

$$
\begin{aligned}
& 2^{681}5^{378} = 1010 = 2^{301}5^{416} \bmod 1019 \\
& \Rightarrow (416 - 378)k = 681 - 301 \\
& \Rightarrow k = 10
\end{aligned}
$$

7.2.4 Parallelizing Pollard's Rho Algorithm

The problem of the traditional Pollard's Rho algorithm is that, due to the serial nature of the random walks, it cannot be efficiently computed in parallel. A solution was proposed by van Oorschot and Wiener in 1999 (van Oorschot and Wiener [1999]). Instead of having two random walks, each processor computes its own individual search path. The walk continues until a point with a certain distinguishing property is found. This point is stored in a central database and the processor starts a new random walk from a random starting point.

The goal is to find two paths with different starting points that end in the same distinguished point. If two such paths are found then a collision is detected and the discrete logarithm can be solved. This process is depicted in Figure 7.4. The paths of Processor w_3 and w_4 collide. Therefore, k can be computed using the exponents of W_4 and U_4.

The advantage of this approach is that a linear speed-up can be achieved. However, a slight overhead is introduced by managing the database and the communication of the separate processors.

The algorithm for multi-processor Pollard's Rho on elliptic curves is described in Algorithm 6.

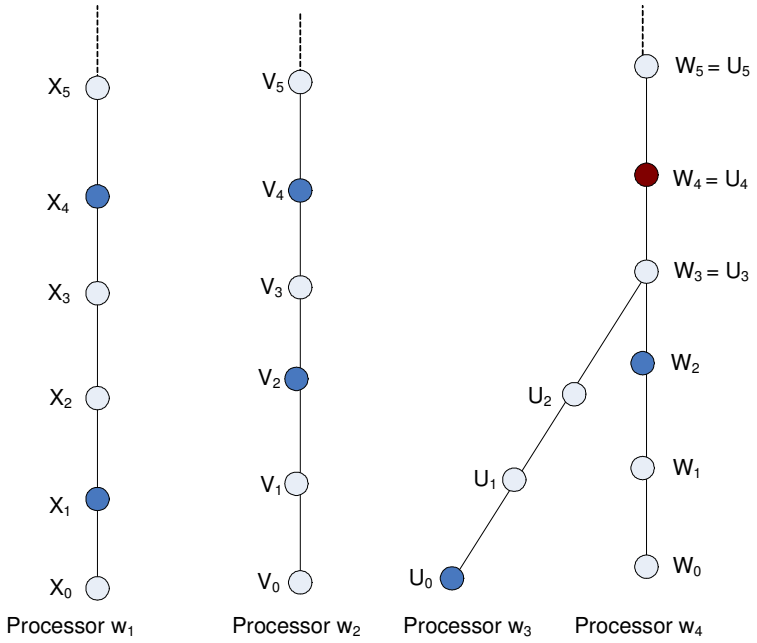


Figure 7.4: Multi-Processor Pollard's Rho

7.3 Tools Supporting Asymmetric Attacks

In this section, implementations and tools to solve both problems are presented.

7.3.1 Tools for Integer Factorization

In the first part, an implementation of the elliptic curve method on the COPACOBANA is described. In the second part, software tools for factorization are discussed.

Elliptic Curve Method on COPACOBANA

In (Zimmermann et al. [2010]), the authors implement the elliptic curve method on the COPACOBANA. The goal is to factorize 100-200-bit integers. DSP blocks are utilized to boost the performance of modular arithmetic (ADD/SUB/MUL). On one Xilinx Virtex 4 SX 35 (XCV4SX35-10) there is enough space for 24 ECM cores. This results in 3,240 factoring operations per second for 202-bit inputs. Nevertheless, the results are still inferior to results obtained on a GPU. With a Nvidia GTX580, it is possible to achieve 171,456 Operations per second for 192-bit inputs.

Software Tools for Factorization

To use algorithms which factorize integers (Pollard's Rho, ECM, Quadratic Sieve), several algebraic mathematical packages can be used. The most popular ones are Magma and Maple. For the elliptic curve method, there exists an implementation using GMP (`http://ecm.gforge.inria.fr/`, `http://eecm.cr.yp.to/`). An implementation of the number field sieve can be found here: `http://cado-nfs.gforge.inria.fr/`. Additionally, there is an implementation of the general number field sieve available online at `http://www.math.ttu.edu/~cmonico/software/ggnfs/`.

Algorithm 6: Multi-Processor Pollard's Rho (Hankerson et al. [2004])

Input: $P \in E(\mathbb{F}_p), n = ord(P), Q \in \langle P \rangle$
Output: The discrete logarithm $l = log_P(Q)$

Select the size s of a finite set with random points
Select a partitioning function $g : \langle P \rangle \rightarrow \{0, 2, ..., s-1\}$
Select a set D out of $\langle P \rangle$ satisfying the distinguished point property
for $i = 0$ to $s-1$ **do**
  Select random coefficients $a_i, b_i \in_R [1, ..., n-1]$
  Compute i-th random point $R_i \leftarrow a_i P + b_i Q$
end for
for each parallel processor **do**
  Select starting coefficient randomly $c, d \in_R [1, ..., n-1]$
  Compute a starting point $X \leftarrow cP + dQ$
  repeat
    if $X \in D$ is a distinguished point **then**
      Send (c, d, X) to the central server
    end if
    Compute partition of current point $i = g(X)$
    Compute next point $X \leftarrow X + R_i; c \leftarrow c + a_i \bmod n; d \leftarrow d + b_i \bmod n$
  until a collision in two points was detected on the server
end for
Let the two colliding triples in point Y be (c_1, d_1, Y) and (c_2, d_2, Y)
if $c_1 = c_2$ **then**
  return failure
else
  Compute $l \leftarrow (c_1 - c_2)(d_2 - d_1)^{-1} \bmod n$
  return l
end if

7.3.2 Tools for Discrete Logarithms

In the first part, implementations of the parallel Pollard's Rho are described. In the second part, software tools for discrete logarithm solving are discussed.

Implementations of Parallel Pollard's Rho

In (Güneysu et al. [2008]), the central server was implemented in software. It was responsible for administrative tasks and storing the distinguished points. The point processors are hardware based and communicate with the central server. Specialized hardware is very suited for this kind of work. A large array of cheap FPGAs or ASICs can be used to increase the efficiency heavily. The basic structure is shown in Figure 7.5.

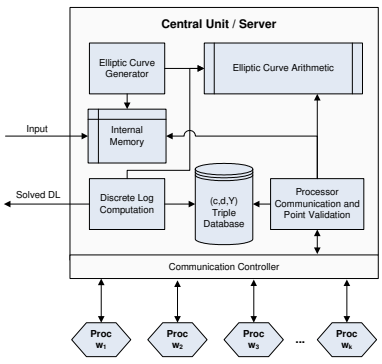


Figure 7.5: Multi-Processor Pollard's Rho

The benchmarks for several FPGA implementation on a **Spartan-3** can be found in Table 7.8.

Table 7.8: Synthesis Results on SPARTAN-3 XC3S1000 FPGA.

Bit size k	# Cores	Device Usage	Max Freq.	Time per OP	Pts/sec per Core	Pts/sec per FPGA
160	2	83%	40.0 MHz	21.4 μs	46.800	**93.600**
128	3	98%	40.1 MHz	17.3 μs	57.800	**173.600**
96	4	98%	44.3 MHz	12.1 μs	82.700	**331.600**
80	4	88%	50.9 MHz	8.94 μs	111.900	**447.600**
64	5	88%	52.0 MHz	7.21 μs	138.600	**693.600**

A different approach is to use multiple Playstation 3 instead of FPGA's. In (Bos et al. [2009]), Pollard's Rho is implemented on multiple PS3. The results are presented in Table 7.9.

Table 7.9: Iterations per second for 96 and 128 bits.

	96 bits	128 bits
COPACOBANA	$4.0 \cdot 10^7$	$2.1 \cdot 10^7$
+ Moore's law	$7.9 \cdot 10^7$	$4.2 \cdot 10^7$
+ Negation map	$1.1 \cdot 10^8$	$4.9 \cdot 10^7$
PS3	$4.2 \cdot 10^7$	$4.2 \cdot 10^7$
33 PS3	$1.4 \cdot 10^9$	$1.4 \cdot 10^9$

Software Tools for Discrete Logarithms

To use algorithms which solve discrete logarithms (Pollard's Rho, Index Calculus), several algebraic mathematical packages can be used. The most popular ones are Magma and Maple. Additionally, an implementation of Index Calculus by Chris Studholme is available online (`http://www.cs.toronto.edu/~cvs/dlog/`).

7.4 Lessons Learned

The following items have been discussed as part of this chapter:

- Factorization and discrete logarithm are both important mathematical problems in cryptography.
- The parameter choice (size of N or group) is important for the security of the cryptosystem.
- The best known attacks on the factorization problem is the General Number Field Sieve (GNFS).
- The best known attacks on discrete logarithms over elliptic curves is the Pollard-Rho algorithm.

Indexes

I. Figures

II. Examples

III. Definitions

IV. Theorems

V. Tables

VI. Bibliography

Open Office - The Free and Open Productivity Suite, Nov 2011. http://www.openoffice.org/.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560.

R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005.

Georg Becker, Francesco Regazzoni, Christof Paar, and Wayne Burleson. Stealthy dopant-level hardware trojans. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 197–214. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40348-4. doi: 10.1007/978-3-642-40349-1_12. URL http://dx.doi.org/10.1007/978-3-642-40349-1_12.

Kostas Bimpikis and Ragesh Jaiswal. Modern factoring algorithms. *University of California, San Diego*, 2005.

M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener. Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security. Technical report, Report of ad hoc panel of cryptographers and computer scientists, Jan. 1996.

Guy E. Blelloch and Bruce M. Maggs. *Parallel Algorithms*. Chapman & Hall/CRC, 2004.

Johannes Blömer. Skript zur vorlesung: Komplexitätstheorie. Technical report, Uni Paderborn, 2006.

Ruud Bolle, Jonathan Connell, Sharanthchandra Pankanti, Nalini Ratha, and Andrew Senior. *Guide to Biometrics*. SpringerVerlag, 2003. ISBN 0387400893.

Joppe W Bos, Marcelo E Kaihara, and Peter L Montgomery. Pollard rho on the playstation 3. In *Workshop record of SHARCS*, volume 9, pages 35–50. Citeseer, 2009.

Uwe Brinkschulte and Theo Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer-Verlag, 2007. ISBN 978-3-540-46801-1.

S.C. Coutinho. *The mathematics of ciphers: number theory and RSA cryptography*. AK Peters, Ltd., 1999. ISBN 978-1568810829.

J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002. ISBN 3-540-42580-2.

M. Daum and S. Licks. Attacking hash functions by poisoned messages, the story of alice and her boss. http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/rump_ec05.pdf, 2005. [Online; accessed 04-July-2014].

ECRYPT II - European Network of Excellence in Cryptology II. Ecrypt ii yearly report on algorithms and keysizes. Technical report, (2011-2012).

M. Sevens et al. Attacking X.509 Certificates with PS3 Clusters. http://www.win.tue.nl/hashclash/rogue-ca/, 2008. [Online; accessed 04-July-2014].

Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA*, pages 140–150, 1983.

M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9): 948–960, Sept 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

Max Gebhardt, Georg Illies, and Werner Schindler. A note on the practical value of single hash collisions for special file formats. In Jana Dittmann, editor, *Sicherheit*, volume 77 of *LNI*, pages 333–344. GI, 2006. ISBN 3-88579-171-4.

Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2006.41. http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf.

Tim Güneysu, Christof Paar, and Jan Pelzl. Special-purpose hardware for solving the elliptic curve discrete logarithm problem. *TRETS*, 1(2), 2008.

D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, 2004.

B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sept. 2000. `http://tools.ietf.org/html/rfc2898`.

John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006. ISBN 3-540-34546-9.

John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption, 5th International Workshop, FSE '98, Paris, France, March 23-25, 1998, Proceedings*, volume 1372 of *Lecture Notes in Computer Science*, pages 168–188. Springer, 1998. doi: 10.1007/3-540-69710-1_12.

Auguste Kerckhoff. La cryptographie militaire. *Journal des sciences militaires*, 9:161–191, 1883.

Lars Knudsen and Matthew Robshaw. *The block cipher companion*. Information security and cryptography. Springer, Heidelberg, London, 2011. ISBN 978-3-642-17341-7. URL `http://opac.inria.fr/record=b1133847`. AU@.

Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

Donald E. Knuth. *The Art of Computer Programming. 2. Seminumerical Algorithms*. Addison-Wesley, 3. auflage edition, 1997. ISBN 0-201-89684-2.

N. Koblitz. *A course in Number Theory and Cryptography*. Springer, 1994. ISBN 978-0387942933.

Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201648652.

Arjen K. Lenstra and Hendrik W. Lenstra Jr. Algorithms in number theory. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 673–716. 1990.

A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

Ondrej Mikle. Practical attacks on digital signatures using md5 message digest. *IACR Cryptology ePrint Archive*, 2004:356, 2004.

National Institute for Standards and Technology (NIST). FIPS SP 800-63-1: Electronic Authentication Guideline , 2011.

National Institute for Standards and Technology (NIST). FIPS PUB 180-4: Digital Signature Standard (DSS), 2013.

Phong Q. Nguyen. Public-key cryptanalysis. *Recent Trends in Cryptography in Contemporary Mathematics series (AMS-RSME)*, 2008.

Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010. ISBN 978-3-642-04100-6.

Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

Carl Pomerance. A tale of two sieves. *Notices Amer. Math. Soc*, 43:1473–1485, 1996.

RegExLib. Regular Expression Cheat Sheet for RegExLib. `http://regexlib.com/CheatSheet.aspx`. [Online; accessed 02-June-2014].

Michael S. Schlansker and B. Ramakrishna Rau. Epic: Explicititly parallel instruction computing. *IEEE Computer*, 33(2):37–45, 2000.

C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001. ISSN 1559-1662. doi: 10.1145/584091.584093. URL `http://doi.acm.org/10.1145/584091.584093`.

Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

Sergei Skorobogatov. Flash memory bumping attacks. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems (CHES 2010)*, volume 6225 of *Lecture Notes in Computer Science*, pages 158–172. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15030-2. doi: 10.1007/978-3-642-15031-9_11. URL `http://dx.doi.org/10.1007/978-3-642-15031-9_11`.

Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009. ISBN 978-3-642-03355-1.

Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-prefix collisions for md5 and applications. *IJACT*, 2(4):322–359, 2012.

TrueCrypt. TrueCrypt - Free Open-Source On-The-Fly Encryption, Nov 2011. `http://www.truecrypt.org/`.

Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

Klaus Wüst. *Mikroprozessortechnik*. Vieweg+Teubner, 2009. ISBN 978-3-8348-0461-7.

Ralf Zimmermann, Tim Güneysu, and Christof Paar. High-performance integer factoring with reconfigurable devices. In *FPL*, pages 83–88. IEEE, 2010.